目录



Rust 程序设计语言（第一版） 简体中文版

本文档为 *The Rust Programming Language* 的中文翻译。
欢迎在 GitHub 上为本文档做出贡献。

关于版本

本书是 Rust 程序设计语言（第一版）的翻译

Rust 程序设计语言（第二版）还在施工中，不过大体已经完成。

Rust 程序设计语言（第二版）官方仓库：https://github.com/rust-lang/book

Rust 程序设计语言（第二版）翻译仓库：https://github.com/KaiserY/trpl-zh-cn

Rust 程序设计语言（第二版）翻译阅读：https://kaisery.github.io/trpl-zh-cn

Rust 版本

1.17.0

相关资源

- GitHub: https://github.com/KaiserY/rust-book-chinese
- GitBook: https://www.gitbook.com/book/kaisery/rust-book-chinese
- Rust 中文社区: https://rust-china.org
- QQ 群: 144605258
- QQ 2 群： 303838735
- 聊天频道：https://chat.rust-china.org/

贡献者

惯例排名不分先后

- aakloxu
- angusdwhite
- armink
- ARwMq9b6
- BingCheung
- Bluek404
- davidwudv
- hczhcz
- hltj
- honorabrutroll
- hy0kl
- imxiaozhi
- iovxw
- JaySon-Huang
- KaiserY
- KaleoCheng
- kenshinji
- kimw
- leqinotes
- linjx
- liubin
- liuchong
- liuzhe0223
- llluiop
- loulihto
- LuoZijun
- mapx
- NemoAlex
- newbienewbie
- opticaline
- peng1999

- qiuyesuifeng
- quxiaolong1504
- rod-lin
- tinunkai
- t123yh
- ustcscgy
- vra
- weaming
- wzv5
- yqylovy
- zhukunqian
- ziqin
- zzh1996
- 1989car

Rust 编程语言

> README.md
> commit 8dc59a54391b8ce44b25659c7c1f8dcde3685bff

欢迎阅读！这本书将教会你使用Rust 编程语言。Rust 是一个系统编程语言，它注重于三个方面：安全，速度和并发性。为了实现这些目标，它没有采用垃圾回收机制（GC）。这让它在其它语言并不擅长的场景中大展身手：嵌入到其它语言中、在特定的时间和空间要求下编程、编写例如设备驱动和操作系统这样的底层代码。它通过一系列不产生运行时开销的编译时安全检查来提升目前语言所关注的领域，同时消除一切数据竞争。Rust 还致力于实现“零开销抽象”，虽然有些抽象看起来更像一个高级语言的特性。即便如此，你仍然可以使用 Rust 来做一些底层的精准控制。

《Rust编程语言》分为数章。本介绍是第一章。之后是：

- 准备 - 为你的电脑安装 Rust 开发环境。
- 教程：猜测游戏 - 通过一个小项目体验 Rust。
- 语法和语义 - Rust 各个部分，被拆分成小的部分讲解。
- 高效 Rust - 编写优秀 Rust 代码的高级内容。
- Rust 开发版 - 还未出现在稳定版本中的最新功能。（译者注：本章节在现版本 1.17.0 中已经去掉，暂时保留，不具有时效性）
- 词汇表 - 书中使用的术语的参考。
- 参考文献 - 影响过 Rust 的文献，关于 Rust 的论文。

贡献

生成这本书（英文版）的源文件可以在 GitHub 上找到。

准备

> getting-started.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

本书的第一部分将带领大家了解 Rust 及其工具。在安装 Rust 之后，我们将开始编写经典的“Hello World”程序。最后将介绍 Cargo —— Rust 的构建系统以及包管理器。

我们将会展示很多使用终端的命令，并且这些行都以 `$` 开头。你并不需要真正输入 `$`，在这里它们代表每行指令的开头。你会在网上看到很多使用这个惯例的教程和例子：`$` 代表常规用户运行的命令，`#` 代表以管理员身份运行的命令。

安装 Rust

使用 Rust 的第一步是安装它。总的来说，你需要联网执行本部分的命令，因为我们要从网上下载 Rust。

Rust 编译器可以运行和编译在许多平台上，不过在 x86 和 x86-64 CPU 构架的 Linux、Mac 和 Windows 平台上的支持是最好的。对于这些和其他一些平台提供官方构建的 Rust 编译器和标准库。关于官方平台支持的全部细节请查看这个网站。

安装 Rust

在 Unix 类系统如 Linux 和 macOS 上，你只需打开终端并输入：

```
$ curl https://sh.rustup.rs -sSf | sh
```

这样会下载一个脚本并开始安装。如果一切顺利，你将会看到：

```
Rust is installed now. Great!
```

在 Windows 上安装也同样简单：下载并运行 rustup-init.exe。其会在终端中开始安装并在成功时显示以上信息。

如需其他安装选项和信息，请访问 Rust 官网的 install 页面。

卸载

卸载 Rust 跟安装它一样容易：

```
$ rustup self uninstall
```

疑难解答

安装完 Rust 以后，我们可以打开 shell，并输入：

```
$ rustc --version
```

你应该能看到版本号、提交的 hash 值和提交时间。

若如是，则 Rust 已成功安装！恭喜你！

若无效，这可能意味着 `PATH` 环境变量并没有包含 Cargo 可执行程序的路径，其在类 Unix 系统下是 `~/.cargo/bin` ，在 Windows 下是 `%USERPROFILE%\.cargo\bin` 。这是存放 Rust 开发工具的路径，绝大多数 Rust 程序员将它放在 `PATH` 系统变量中，以便于在命令行运行 `rustc` 。根据操作系统或命令行 shell 的不同，以及安装过程的 bug，你可能需要重启 shell，注销系统，或者为你的操作环境手动配置合适的 `PATH` 。

Rust 并没有自己的连接器，所以你需要自己装一个。做法因特定的系统而有所不同。对于 Linux 系统，Rust 会尝试调用 `cc` 进行连接。对于 `windows-msvc` （在 Windows 上使用 Microsoft Visual Studio 构建的 Rust），则需要安装Microsoft Visual C++ Build Tools。其并不需要位于 `%PATH%` 中，因为 `rustc` 会自动找到他们。一般来说，如果你的连接器位于一个不常见的位置，你需要调用 `rustc linker=/path/to/cc` ，其中 `/path/to/cc` 指向连接器的路径。

如果还是搞不定，我们有许多可以获取帮助的地方。最简单的是 irc.mozilla.org 上的 IRC 频道 #rust-beginners 和供一般讨论之用的 #rust，我们可以使用 Mibbit 访问之。然后我们就可以和其他能提供帮助的 Rustacean（我们这些人自称的愚蠢绰号）聊天了。其它给力的资源包括用户论坛和Stack Overflow。

安装程序也会在本地安装一份文档拷贝，你可以离线阅读它们。只需输入 `rustup doc` 即可！

Hello, world!

现在你已经安装好了 Rust，我们将帮助你编写你的第一个 Rust 程序。当学习一门新语言的时候编写一个在屏幕上打印 “Hello, world!” 文本的小程序是一个传统，而在这一部分，我们将遵循这个传统。

以这样一个简单的程序开始的好处是你可以快速的确认你的编译器已正确安装，并可以正常工作。在屏幕上打印信息也是一个非常常见的操作，所以早点实践一下是有好处的。

> 注意：本书假设你熟悉基本的命令行操作。Rust 本身并不对你的编辑器，工具和你的代码存放在何处有什么特定的要求，所以如果你比起命令行更喜欢 IDE，这里也有一个选择。你可能想要试试SolidOak，它专为 Rust 而设计。在 Rust 社区里有许许多多正在开发中的 IDE 插件。Rust 团队也发布了不同编辑器的插件。 配置编辑器或 IDE 已超出本教程的范畴，所以请查看你特定设置的文档。

创建项目文件

首先，创建一个文件来编写 Rust 代码。Rust 并不关心你的代码存放在哪里，不过在本书中，我们建议在你的 home 目录创建一个存放项目的目录，并把你的所有项目放在这。打开一个终端并输入如下命令来为这个项目创建一个文件夹：

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

> 如果你使用 Windows 并且没有用 PowerShell， `~` 可能不指向你的用户目录。可以查询所使用的 Shell 的相关文档以获取更多信息。

编写并运行 Rust 程序

我们需要为我们的 Rust 程序创建一个源文件。Rust 代码文件总是使用 `.rs` 后缀，并且如果我们用的 Rust 文件名由多个单词组成，我们使用下划线分隔它们；例如，使用*my_program.rs*而不是*myprogram.rs*。

现在，创建一个名叫*main.rs*的新文件。打开并输入如下代码：

```
fn main() {
    println!("Hello, world!");
}
```

保存文件，并回到你的命令行窗口。在 Linux 或 OSX 上，输入如下命令：

```
$ rustc main.rs
$ ./main
Hello, world!
```

在 Windows 下，使用 `main.exe` 而不是 `main` 。不管你使用何种系统，你应该在终端看到 `Hello, world!` 字符串。如果你做到了，那么恭喜你！你已经正式编写了一个 Rust 程序。你是一名 Rust 程序员了！欢迎入坑。

分析 Rust 程序

现在，让我们回过头来仔细看看你的“Hello, world!”程序到底发生了什么。这里是拼图的第一片：

```
fn main() {

}
```

这几行定义了一个 Rust 函数。 `main` 函数是特殊的：这是所有 Rust 程序的开始。第一行表示“定义一个叫 `main` 的函数，没有参数也没有返回值。”如果有参数的话，它们应该出现在括号（ `(` 和 `)` ）中。因为并没有返回值，所以我们可以省略整个返回值类型。

同时注意函数体被包裹在大括号（ `{` 和 `}` ）中。Rust 要求所有函数体都位于其中。将前一个大括号与函数声明置于一行，并留有一个空格被认为是一个好的代码风格。

在 `main()` 函数中：

```
    println!("Hello, world!");
```

这行代码做了这个小程序的所有工作：它在屏幕上打印文本。这有很多重要的细节。第一个是使用 4 个空格缩进，而不是制表符。

第二个重要的部分是 `println!()` 这一行。这是一个 Rust 宏，是 Rust 元编程的关键所在。相反如果我们调用一个函数的话，它应该看起来像这样： `println()` （没有 !）。我们将在后面更加详细的讨论 Rust 宏，不过现在你只需记住当看到符号 `!` 的时候，就代表调用了一个宏而不是一个普通的函数。

接下来， `"Hello, world!"` 是一个字符串。在一门系统级编程语言中，字符串是一个复杂得令人惊讶的话题。这是一个静态分配的字符串。这个语句将这个字符串作为参数传递给 `println!` 宏，这个宏负责在屏幕（控制台）上打印字符串。是不是很简单啊(⊙o⊙)

这一行以一个分号结尾（ `;` ）。Rust是一门面向表达式的语言，也就是说大部分语句都是表达式。 `;` 表示一个表达式的结束，另一个新表达式的开始。大部分 Rust 代码行以 `;` 结尾。

编译和运行是两个步骤

在“编写并运行一个 Rust 程序”，我们展示了如何运行一个新创建的程序。现在我们将拆分并检查每一个操作。

在运行一个 Rust 程序之前，你必须编译它。你可以输入 `rustc` 命令来使用 Rust 编译器并像这样传递你源文件的名字：

```
$ rustc main.rs
```

如果你来自 C 或 C++ 背景，你会发现这与 `gcc` 和 `clang` 类似。编译成功后，Rust 应该会输出一个二进制可执行文件，在 Linux 或 OSX 下在shell 中通过如下 `ls` 命令你可以看到它：

```
$ ls
main  main.rs
```

在 Windows 下，输入：

```
$ dir
main.exe
main.rs
```

这表示我们有两个文件：`.rs` 后缀的源文件，和可执行文件（在 Windows下是 `main.exe` ，其它平台是 `main` ）。这里我们剩下的操作就只有运行 `main` 或 `main.exe` 文件了，像这样：

```
$ ./main  # or .\main.exe on Windows
```

如果 `main.rs` 是我们的“Hello, world!”程序，它将会在你的终端上打印 `Hello, world!` 。

来自 Ruby、Python 或 JavaScript 这样的动态类型语言背景的同学，可能不太习惯这样将编译和执行分开。Rust 是一种 预编译语言（*ahead-of-time compiled language*），程序编译好后，把它给任何人，他们都不需要安装 Rust 就可运行。如果你给他们一个 `.rb` ， `.py` 或 `.js` 文件，他们需要先分别安装 Ruby，Python，JavaScript 实现，不过你只需要一句命令就可以编译和执行你的程序。这一切都是语言设计的权衡取舍。

仅仅使用 `rustc` 编译简单程序是没问题的，不过随着你的项目的增长，你将想要能够控制你项目拥有的所有选项，并易于分享你的代码给别人或别的项目。接下来，我们将介绍一个叫做 Cargo 的工具，它将帮助你编写现实生活中的 Rust 程序。

Hello, Cargo!

Cargo 是 Rust 的构建系统和包管理工具，同时 Rustacean 们使用 Cargo 来管理它们的 Rust 项目。Cargo 负责三个工作：构建你的代码，下载你代码依赖的库并编译这些库。我们把你代码需要的库叫做“依赖（dependencies）”因为你的代码依赖他们。

最简单的 Rust 程序并没有任何依赖，所以目前我们只使用它的第一部分功能。随着你编写更加复杂的 Rust 程序，你会想要添加依赖，那么如果你使用 Cargo 开始的话，这将会变得简单许多。

因为绝大部分 Rust 项目使用 Cargo，本书接下来的部分将假设你使用它。如果你使用官方安装包的话，Rust 自带 Cargo。如果你使用其他方式安装 Rust 的话，你可以在终端输入如下命令检查你是否安装了 Cargo：

```
$ cargo --version
```

如果你看到了版本号，一切 OK！如果你一个类似“ `command not found` ”的错误，那么你应该去查看你安装 Rust 的系统的相关文档，来确定 Cargo 是否需要单独安装。

转换到 Cargo

让我们将 Hello World 程序迁移至 Cargo。为了 Cargo 化一个项目，需要做三件事：

1. 将源文件放到正确的目录
2. 删除旧的可执行文件（Windows下是 `main.exe` ，其他平台是 `main` ）。
3. 创建一个 Cargo 配置文件

让我们开始吧！

创建源文件目录并移除旧的可执行文件

首先，回到你的终端，移动到你的 `hello_world` 目录，并输入如下命令：

```
$ mkdir src
$ mv main.rs src/main.rs # or 'move main.rs src/main.rs' on Windows
$ rm main  # or 'del main.exe' on Windows
```

Cargo 期望源文件位于 src 目录，所以先做这个。这样将项目顶级目录（在这里，是 hello_world）留给 README，license 信息和其他跟代码无关的文件。这样，Cargo 帮助你保持项目干净整洁。一切井井有条。

现在，移动 `main.rs` 到 `src` 目录，并删除你用 `rustc` 创建的编译过的文件。像往常一样，如果你使用 Windows，用 `main.exe` 代替 `main` 。

例子中我们继续使用 `main.rs` 作为源文件名是因为它创建了一个可执行文件。如果你想要创建一个库文件，使用 `lib.rs` 作为文件名。Cargo 使用这个约定来正确编译你的项目，不过如果你愿意的话也可以覆盖它。

创建配置文件

下一步，在 `hello_world` 目录创建一个文件，叫做 `Cargo.toml` 。

确保 `Cargo.toml` 的 `C` 是大写的，否则 Cargo 不知道如何处理配置文件。

这个文件使用TOML（Tom's Obvious, Minimal Language）格式。 TOML 类似于 INI，不过有一些额外的改进之处，并且被用作 Cargo 的配置文件。

在这个文件中，输入如下信息：

```
[package]

name = "hello_world"
version = "0.0.1"
authors = [ "Your name <you@example.com>" ]
```

第一行的 `[package]` 表明下面的语句用来配置一个包。随着我们在这个文件增加更多的信息，我们会增加其他部分，不过现在，我们只有包配置。

另外三行设置了 Cargo 编译你的程序所需要知道的三个配置：包的名字，版本，和作者。

当你在 `Cargo.toml` 中添加完这些信息后，保存它以完成配置文件的创建。

构建并运行 Cargo 项目

当 `Cargo.toml` 文件位于项目的根目录时，我们就准备好可以构建并运行 Hello World 程序了！为此，我们输入如下命令：

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/
hello_world)
$ ./target/debug/hello_world
Hello, world!
```

如果一切顺利，你应该再次看到 `Hello, world!` 出现在终端里。

你刚刚用 `cargo build` 构建了一个程序并用 `./target/debug/hello_world` 运行了它，不过你也可以用如下的一步操作 `cargo run` 来完成这两步操作：

```
$ cargo run
     Running `target/debug/hello_world`
Hello, world!
```

`run` 命令在你需要快速迭代项目时显得很有用。

注意这个例子并没有重新构建项目。Cargo 发现文件并没有被修改，所以它只是运行了二进制文件。如果你修改了源文件，Cargo 会在运行前重新构建项目，这样你将看到像这样的输出：

```
$ cargo run
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/
hello_world)
     Running `target/debug/hello_world`
Hello, world!
```

Cargo 检查任何项目文件是否被修改，并且只会在你上次构建后修改了他们才重新构建。

对于简单的项目，Cargo 并不比使用 `rustc` 要好多少，不过将来它会变得有用。这在你开始使用 crate 时显得尤为正确；（crate）在其他语言中有“库（library）”或“包（package）”这样的同义词。对于包含多个 crate 的项目，让 Cargo

来协调构建将会轻松很多。有了 Cargo，你可以运行 `cargo build` ，而一切将有条不紊的运行。

发布构建

当你的项目准备好发布了，可以使用 `cargo build --release` 来优化编译项目。这些优化可以让 Rust 代码运行的更快，不过启用他们会让程序花更长的时间编译。这也是为何这是两种不同的配置，一个为了开发，另一个构建提供给用户的最终程序。

那个 `Cargo.lock` 是什么？

运行这个命令同时也会让 Cargo 创建一个叫做 `Cargo.lock` 的文件，它看起来像这样：

```
[root]
name = "hello_world"
version = "0.0.1"
```

Cargo 用 `Cargo.lock` 文件跟踪你程序的依赖。这里是 Hello World 项目的 `Cargo.lock` 文件。这个项目并没有依赖，所以内容有一点稀少。事实上，你自己甚至都不需要碰这个文件；仅仅让 Cargo 处理它就行了。

就是这样！如果你一路跟过来了，你应该已经成功使用 Cargo 构建了 `hello_world` 。

虽然这个项目很简单，现在它使用了很多在你余下的 Rust 生涯中将会用到的实际的工具。事实上，你可以期望使用如下命令的变体开始所有的 Rust 项目：

```
$ git clone someurl.com/foo
$ cd foo
$ cargo build
```

创建新 Cargo 项目的简单方法

你并不需要每次都过一遍上面的操作来开始一个新的项目！Cargo 可以快速创建一个骨架项目目录这样你就可以立即开始开发了。

用 Cargo 来开始一个新项目，在命令行输入 `cargo new` ：

```
$ cargo new hello_world --bin
```

这个命令传递了 `--bin` 参数因为我们的目标是直接创建一个可执行程序，而不是一个库。可执行文件通常叫做二进制文件（因为它们位于 `/usr/bin` ，如果你使用 Unix 系统的话）。

Cargo 为我们创建了两个文件和一个目录：一个 `Cargo.toml` 和一个包含了 `main.rs` 文件的 `src` 目录。这应该看起来很眼熟，他们正好是我们在之前手动创建的那样。

这些输出是你开始所需要的一切。首先，打开 `Cargo.toml` 。它应该看起来像这样：

```
[package]

name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

不要担心 `[dependencies]` 那一行，之后我们会讲到。

Cargo 已经根据你给出的参数和 `git` 全局配置给出了合理的默认配置。你可能会注意到 Cargo 也把 `hello_world` 目录初始化为了一个 `git` 仓库。

这是应该写入 `src/main.rs` 的代码：

```
fn main() {
    println!("Hello, world!");
}
```

Cargo 已经为你生成了一个“Hello World!”，现在你已经准备好开始撸代码了！

> 注意：如果你想要查看 Cargo 的详细信息，请查看官方的Cargo 指导，它包含了所有这些功能。

写在最后

本章涵盖了将用于本书余下部分以及你之后 Rust 时光的基础知识。现在你获得了工具，我们将更多地介绍 Rust 语言本身。

接下来你有两个选择：在 “学习 Rust” 中深入研究一个项目，或者自下而上地学习 “语法和语义”。来自系统级编程语言的同学，你们可能倾向于选择 “学习 Rust”，而来自动态编程语言的同学，请根据自己的喜好来选择吧。人各有别，适合自己的才是最好的。

猜猜看

> guessing-game.md
> commit 602fa7c6eba1b0fbeca0847fe295d9137e9a0243

让我们学习一些 Rust！作为第一个项目，我们来实现一个经典新手编程问题：猜猜看游戏。它是这么工作的：程序将会随机生成一个 1 到 100 之间的随机数。它接着会提示猜一个数。当我们猜了一个数之后，它会告诉我们是太大了还是太小了。猜对了，它会祝贺我们。听起来如何？

准备

让我们准备一个新项目。进入到项目目录。还记得之前如何创建 `hello_world` 的项目目录和 `Cargo.toml` 文件的吗？Cargo 有一个命令来做这些。让我们试试：

```
$ cd ~/projects
$ cargo new guessing_game --bin
     Created binary (application) `guessing_game` project
$ cd guessing_game
```

我们将项目名字传递给 `cargo new` ，然后用了 `--bin` 标记，因为要创建一个二进制文件，而不是一个库文件。

查看生成的 `Cargo.toml` 文件：

```
[package]

name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
```

Cargo 从系统环境变量中获取这些信息。如果这不对，赶紧修改它。

最后，Cargo 为我们生成了一个“Hello, world!”。查看 `src/main.rs` 文件：

```
fn main() {
    println!("Hello, world!");
}
```

让我们编译 Cargo 为我们生成的项目：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.53 s
ecs
```

很好！再次打开你的 `src/main.rs` 文件。我们会将所有代码写在这个文件里。稍后我们会讲到多文件项目。

还记得上一章节讲到的 `run` 命令吗？让我们再次试试它：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 se
cs
     Running `target/debug/guessing_game`
Hello, world!
```

很好！我们的小游戏恰恰是 `run` 命令大显身手的这类程序：我们需要在进行下一步之前快速测试每次迭代。

处理一次猜测

让我们开始吧！我们需要做的第一件事是让我们的玩家输入一个猜测。把这些放入你的 `src/main.rs`：

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

这有好多东西！让我们一点一点地过一遍。

```
use std::io;
```

我们需要获取用户输入，并接着打印结果作为输出。为此，我们需要标准库的 `io` 库。Rust 为所有程序只导入了很少一些东西，'prelude'。如果它不在预先导入中，你将不得不直接 `use` 它。这还有第二个"prelude", `io` prelude，它也起到了类似的作用：你引入它，它引入一系列拥有的 IO 相关的库。

```
fn main() {
```

就像你之前见过的，`main()` 是你程序的入口点。`fn` 语法声明了一个新函数，`()` 表明这里没有参数，而 `{` 开始了函数体。因为不包含返回类型，它假设是 `()`，一个空的元组。

```
    println!("Guess the number!");

    println!("Please input your guess.");
```

我们之前学过 `println!()` 是一个在屏幕上打印字符串的宏。

```
    let mut guess = String::new();
```

现在我们遇到有意思的东西了！这一小行有很多内容。第一个我们需要注意到的是let语句，它用来创建“变量绑定”。它使用这个形式：

```
let foo = bar;
```

这会创建一个叫做 `foo` 的新绑定，并绑定它到 `bar` 这个值上。在很多语言中，这叫做一个“变量"，不过 Rust 的变量绑定暗藏玄机。

例如，它们默认是不可变的。这时为什么我们的例子使用了 `mut` ：它让一个绑定可变，而不是不可变。 `let` 并不从左手边获取一个名字，事实上它接受一个模式（pattern）。我们会在后面更多的使用模式。现在它使用起来非常简单：

```
let foo = 5; // immutable.
let mut bar = 5; // mutable
```

噢，同时 `//` 会开始一个注释，直到这行的末尾。Rust 忽略注释中的任何内容。

那么现在我们知道了 `let mut guess` 会引入一个叫做 `guess` 的可变绑定，不过我们也必须看看 `=` 的右侧所绑定的内容： `String::new()` 。

`String` 是一个字符串类型，由标准库提供。String是一个可增长的，UTF-8编码的文本。

`::new()` 语法用了 `::` 因为它是一个特定类型的”关联函数“。这就是说，它与 `String` 自身关联，而不是与一个特定的 `String` 实例关联。一些语言管这叫一个“静态方法”。

这个函数叫做 `new()` ，因为它创建了一个新的，空的 `String` 。你会在很多类型上找到 `new()` 函数，因为它是创建一些类型新值的通常名称。

让我们继续：

```
    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");
```

这稍微有点多！让我们一点一点来。第一行有两部分。这是第一部分：

```
io::stdin()
```

还记得我们如何在程序的第一行 `use std::io` 的吗？现在我们调用了一个与之相关的函数。如果我们不 `use std::io` ，那么我们就得写成 `std::io::stdin()` 。

这个特殊的函数返回一个指向你终端标准输入的句柄。更具体的，可参考std::io::Stdin。

下一部分将用这个句柄去获取用户输入：

```
.read_line(&mut guess)
```

这里，我们对我们的句柄调用了read_line()方法。"方法"就像关联函数，不过只在一个类型的特定实例上可用，而不是这个类型本身。我们也向 `read_line()` 传递了一个参数： `&mut guess` 。

还记得我们上面怎么绑定 `guess` 的吗？我们说它是可变的。然而， `read_line` 并不接收 `String` 作为一个参数：它接收一个 `&mut String` 。Rust有一个叫做"引用"的功能，它允许你对一片数据有多个引用，用它可以减少拷贝。引用是一个复杂的功能，因为Rust的一个主要卖点就是它如何安全和便捷地使用引用。然而，目前我们还不需要知道很多细节来完成我们的程序。现在，所有我们需要了解的是像 `let` 绑定，引用默认是不可变的。因此，我们需要写成 `&mut guess` ，而不是 `&guess` 。

为什么 `read_line()` 会需要一个字符串的可变引用呢？它的工作是从标准输入获取用户输入，并把它放入一个字符串。所以它用字符串作为参数，为了可以增加输入，它必须是可变的。

不过我们还未完全看完这行代码。虽然它是单独的一行代码，但只是这个单独逻辑代码行的开头部分：

```
        .expect("Failed to read line");
```

当你用 `.foo()` 语法调用一个函数的时候，你可能会引入一个新行符或其它空白。这帮助我们拆分长的行。我们可以这么干：

```
    io::stdin().read_line(&mut guess).expect("failed to read line");
```

不过这样会难以阅读。所以我们把它分开，3 行对应 3 个方法调用。我们已经谈论过了 `read_line()` ，不过 `expect()` 呢？好吧，我们已经提到过 `read_line()` 将用户输入放入我们传递给它的 `&mut String` 中。不过它也返回一个值：在这个例子中，一个io::Result。Rust的标准库中有很多叫做 `Result` 的类型：一个泛型Result，然后是子库的特殊版本，例如 `io::Result` 。

这个 `Result` 类型的作用是编码错误处理信息。 `Result` 类型的值，像任何（其它）类型，有定义在其上的方法。在这个例子中， `io::Result` 有一个expect()方法获取调用它的值，而且如果它不是一个成功的值，panic!并带有你传递给它的信息。这样的 `panic!` 会使我们的程序崩溃，显示（我们传递的）信息。

如果我们去掉这两个函数调用，我们的程序会编译通过，不过我们会得到一个警告：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
warning: unused result which must be used, #[warn(unused_must_use)] on by default
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^

    Finished debug [unoptimized + debuginfo] target(s) in 0.42 secs
```

Rust警告我们我们并未使用 `Result` 的值。这个警告来自 `io::Result` 的一个特殊注解。Rust 尝试告诉你你并未处理一个可能的错误。阻止错误的正确方法是老实编写错误处理。幸运的是，如果我们只是想如果这有一个问题就崩溃的话，我们可

以用这两个小方法。如果我们想从错误中恢复什么的，我们得做点别的，不过我们会把它留给接下来的项目。

这是我们第一个例子仅剩的一行：

```
    println!("You guessed: {}", guess);
}
```

这打印出我们保存输入的字符串。`{}`是一个占位符，所以我们传递`guess`作为一个参数。如果我们有多个`{}`，我们应该传递多个参数：

```
let x = 5;
let y = 10;

println!("x and y: {} and {}", x, y);
```

轻松加愉快。

总而言之，这只是一个观光。我们可以用`cargo run`运行代码：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.44 s
ecs
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

好的！我们的第一部分完成了：我们可以从键盘获取输入，并把它打印回去。

生成一个秘密数字

接下来，我们要生成一个秘密数字。Rust标准库中还未包含随机数功能。然而，Rust 团队确实提供了一个 `rand` crate。一个“包装箱”（crate）是一个 Rust 代码的包。我们已经构建了一个”二进制包装箱“，它是一个可执行文件。 `rand` 是一个”库包装箱“，它包含被认为应该被其它程序使用的代码。

使用外部包装箱是 Cargo 的亮点。在我们使用 `rand` 编写代码之前，我们需要修改我们的 `Cargo.toml` 。打开它，并在末尾增加这几行：

```
[dependencies]

rand = "0.3.0"
```

`Cargo.toml` 的 `[dependencies]` 部分就像 `[package]` 部分：所有之后的东西都是它的一部分，直到下一个部分开始。Cargo使用依赖部分来知晓你用的外部包装箱的依赖，和你要求的版本。在这个例子中，我们用了 `0.3.0` 版本。Cargo理解语义化版本，它是一个编写版本号的标准。像上面只有数字的版本事实上是 `^0.3.0` 的简写，代表“任何兼容 0.3.0 的版本”。如果你只想使用 `0.3.0` 版本，你可以使用 `rand = "=0.3.0"` （注意那两个双引号）。我们也可以指定一个版本范围。Cargo文档包含更多细节。

现在，在不修改任何我们代码的情况下，让我们构建我们的项目：

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-in
dex`
 Downloading rand v0.3.14
 Downloading libc v0.2.17
   Compiling libc v0.2.17
   Compiling rand v0.3.14
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 5.88 s
ecs
```

（当然，你可能会看到不同的版本）

很多新的输出！现在我们有了一个外部依赖，Cargo 从记录中获取了所有东西的最新版本，它们是来自Crates.io的一份拷贝。Crates.io 是 Rust 生态系统中人们发表开源 Rust 项目供他人使用的地方。

在更新了记录后，Cargo 检查我们的 `[dependencies]` 并下载任何我们还没有的东西。在这个例子中，虽然我们只说了我们要依赖 `rand` ，我们也获取了一份 `libc` 的拷贝。这是因为 `rand` 依赖 `libc` 工作。在下载了它们之后，它编译它们，然后接着编译我们的项目。

如果我们再次运行 `cargo build` ，我们会得到不同的输出：

```
$ cargo build
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 se
cs
```

没错，没有输出！Cargo 知道我们的项目被构建了，并且所有它的依赖也被构建了，所以没有理由再做一遍所有这些。没有事情做，它简单地退出了。如果我们再打开 `src/main.rs` ，做一个无所谓的修改，然后接着再保存，我们就会看到一行：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.45 s
ecs
```

所以，我们告诉Cargo我们需要任何 `0.3.x` 版本的 `rand` ，并且因此它获取在本文被编写时的最新版， `v0.3.14` 。不过你瞧瞧当下一周， `v0.3.15` 出来了，带有一个重要的 bug 修改吗？虽然 bug 修改很重要，不过如果 `0.3.15` 版本包含破坏我们代码的回归缺陷（regression）呢？

这个问题的回答是现在你会在你项目目录中找到的 `Cargo.lock` 。当你第一次构建你的项目的时候，Cargo 查明所有符合你的要求的版本，并接着把它们写到了 `Cargo.lock` 文件里。当你在未来构建你的项目的时候，Cargo 会注意到 `Cargo.lock` 的存在，并接着使用指定的版本而不是再次去做查明版本的所有工作。这让你有了一个可重复的自动构建。换句话说，我们会保持在 `0.3.8` 直到我们显式的升级，这对任何使用我们共享的代码的人同样有效，感谢锁文件。

当我们确实想要使用 `v0.3.15` 怎么办？Cargo 有另一个命令，`update`，它代表“忽略锁，搞清楚所有我们指定的最新版本。如果这能工作，将这些版本写入锁文件”。不过，默认，Cargo 只会寻找大于 `0.3.0` 小于 `0.4.0` 的版本。如果你想要移动到 `0.4.x`，我们不得不直接更新 `Cargo.toml` 文件。当我们这么做，下一次我们 `cargo build`，Cargo会更新索引并重新计算我们的 `rand` 要求。

关于Cargo和它的生态系统有很多东西要说，不过眼下，这是我们需要知道的一切。Cargo让重用库变得真正的简单，并且Rustacean们可以编写更小的由很多子包组装成的项目。

让我们真正的使用 `rand`，这是我们的下一步：

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    println!("You guessed: {}", guess);
}
```

先修改第一行。现在它是 `extern crate rand`。因为在 `[dependencies]` 声明了 `rand`，我们可以用 `extern crate` 来让Rust知道我们正在使用它。这也等同于一个 `use rand;`，所以我们可以通过 `rand::` 前缀使用 `rand` 包装箱中的一切。

下一步，我们增加了另一行`use`：`use rand::Rng`。我们一会将要使用一个方法，并且它要求`Rng`在作用域中才能工作。这个基本观点是：方法定义在一些叫做“特性（traits，也有译作特质）”的东西上面，而为了让方法能够工作，需要这个特性位于作用域中。关于更多细节，阅读trait部分。

这里还有两行我们增加的，在中间：

```
    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);
```

我们用`rand::thread_rng()`函数来获取一个随机数生成器的拷贝，它位于我们特定的执行线程的本地。因为`use rand::Rng`了，有一个`gen_range()`方法可用。这个函数获取两个参数，并产生一个位于其间的数字。它包含下限，不过不包含上限，所以需要`1`和`101`来生成一个`1`和`100`之间的数。

第二行仅仅打印出了秘密数字，这在开发程序时做简单测试很有用。不过在最终版本中我们会删除它。在开始就打印出结果就没什么可玩的了！

尝试运行新程序几次：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.55 secs
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

好的！接下来：让我们比较我们的猜测和秘密数字。

比较猜测

现在我们得到了用户输入，让我们比较我们的猜测和随机值。这是我们的下一步，虽然它还不能正常工作：

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

这有一些新东西。第一个是另一个 `use` 。我们带来了一个叫做 `std::cmp::Ordering` 类型到作用域中。接着，底部5行代码使用了它：

```
match guess.cmp(&secret_number) {
    Ordering::Less    => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal   => println!("You win!"),
}
```

`cmp()` 可以在任何能被比较的值上调用，并且它获取你想要比较的值的引用。它返回我们之前 `use` 的 `Ordering` 类型。我们使用一个match语句来决定具体是哪种 `Ordering` 。 `Ordering` 是一个枚举（enum），它看起来像这样：

```
enum Foo {
    Bar,
    Baz,
}
```

通过这个定义，任何 `Foo` 可以是 `Foo::Bar` 或者 `Foo::Baz` 。我们用 `::` 来表明一个特定 `enum` 变量的命名空间。

Ordering枚举有3个可能的变量： `Less` ， `Equal` 和 `Greater` 。 `match` 语句获取类型的值，并让你为每个可能的值创建一个“分支”。因为有 3 种类型的 `Ordering` ，我们有 3 个分支：

```
match guess.cmp(&secret_number) {
    Ordering::Less    => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal   => println!("You win!"),
}
```

如果它是 `Less` ，我们打印 `Too small!` ，如果它是 `Greater` ， `Too big!` ，而如果 `Equal` ， `You win!` 。 `match` 真的非常有用，并且在 Rust 中经常使用。

我确实提到过我们还不能正常运行，虽然。让我们试试：

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^ expected struct `std::st
ring::String`, found integral variable
   |
   = note: expected type `&std::string::String`
   = note:    found type `&{integer}`

error: aborting due to previous error

error: Could not compile `guessing_game`.

To learn more, run the command again with --verbose.
```

噢！这是一个大错误。它的核心是我们有“不匹配的类型”。Rust 有一个强大的静态类型系统。然而，它也有类型推断。当我们写 `let guess = String::new()`，Rust能够推断出 `guess` 应该是一个 `String`，并因此不需要我们写出类型。而我们的 `secret_number`，这有很多类型可以有从 `1` 到 `100` 的值：`i32`，一个 32 位数，或者 `u32`，一个无符号的32位值，或者 `i64`，一个 64 位值。或者其它什么的。目前为止，这并不重要，所以 Rust 默认为 `i32`。然而，这里，Rust 并不知道如何比较 `guess` 和 `secret_number`。它们必须是相同的类型。最终，我们想要我们作为输入读到的 `String` 转换为一个真正的数字类型，来进行比较。我们可以用额外 3 行来搞定它。这是我们的新程序：

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

新的两行是：

```
    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");
```

稍等，我认为我们已经用过了一个 `guess` ？确实，不过 Rust 允许我们用新值“遮盖（shadow）”之前的 `guess` 。这在这种具体的情况中经常被用到， `guess` 开始是一个 `String` ，不过我们想要把它转换为一个 `u32` 。遮盖（Shadowing）让我们重用 `guess` 名字，而不是强迫我们想出两个独特的像 `guess_str` 和 `guess` ，或者别的什么。

我们绑定 `guess` 到一个看起来像我们之前写的表达式：

```
guess.trim().parse()
```

这里， `guess` 引用旧的 `guess` ，那个我们输入用到的 `String` 。 `String` 的 `trim()` 方法会去掉我们字符串开头和结尾的任何空格。这很重要，因为我们不得不按“回车”键来满足 `read_line()` 。这意味着如果输入 `5` 并按回车， `guess` 看起来像这样： `5\n` 。 `\n` 代表“新行”，回车键。 `trim()` 去掉这些，保留 `5` 给我们的字符串。字符串的 `parse()` 方法将字符串解析为一些类型的数字。因为它可以解析多种数字，我们需要给Rust一些提醒作为我们具体想要的数字的类型。因此， `let guess: u32` 。 `guess` 后面的分号（ `:` ）告诉 Rust 我们要标注它的类型。 `u32` 是一个无符号的，32位整型。Rust 有一系列内建数字类型，不过我们选择了 `u32` 。它是一个小正数的默认好选择。

就像 `read_line()` ，我们调用 `parse()` 可能产生一个错误。如果我们的字符串包含 `A%?` 呢？并不能将它们转换成一个数字。为此，我们将做我们在 `read_line()` 时做的相同的事：使用 `expect()` 方法来在这里出现错误时崩溃。

让我们尝试下我们的程序！

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.57 s
ecs
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

很好！你可以看到我甚至在我的猜测前加上了空格，不过它仍然识别出我猜了76。运行这个程序几次，并检测猜测正确的值，和小的值。

现在我们让游戏大体上能玩了，不过我们只能猜一次。让我们增加循环来改变它！

循环

`loop` 关键字给我们一个无限循环。让我们加上它：

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => println!("You win!"),
        }
    }
}
```

并试试看。不过稍等，难道我们仅仅加上一个无限循环吗？是的。记得我们我们关于 `parse()` 的讨论吗？如果我们给出一个非数字回答，明显我们会 `panic!` 并退出：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.58 s
ecs
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!'
```

啊哈！`quit`确实退出了。就像任何其它非数字输入。好吧，这至少不是最差的想法。首先，如果你赢得了游戏，那我们就真的退出它：

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

通过在 `You win!` 后增加 `break` ，我们将在你赢了后退出循环。退出循环也意味着退出程序，因为它是 `main()` 中最后的东西。我们仅仅需要再做一个小修改：当谁输入了一个非数字，我们并不想退出，我们就想忽略它。我们可以这么做：

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

这是改变了的行：

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

这是你如何大体上从“错误就崩溃”移动到“确实处理错误”，通过从 `expect()` 切换到一个 `match` 语句。 `parse()` 返回的 `Result` 就是一个像 `Ordering` 一样的枚举，不过在这个例子中，每个变量有一些数据与之相关： `Ok` 是一个成功，而 `Err` 是一个失败。每一个都包含更多信息：成功的解析为整型，或一个错误类型。在这个例子中，我们 `match` 为 `Ok(num)` ，它设置了 `Ok` 内叫做 `num` 的值，接着在右侧返回它。在 `Err` 的情况下，我们并不关心它是什么类型的错误，所以我们仅仅使用 `_` 而不是一个名字。这忽略错误，并 `continue` 造成我们进行 `loop` 的下一次迭代。

现在应该搞定了！试试看：

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/gue
ssing_game)
    Finished debug [unoptimized + debuginfo] target(s) in 0.57 s
ecs
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

狂拽炫酷！通过一个最后的修改，我们就完成了猜猜看游戏。你能想到它是什么吗？对了，我们并不想打印出秘密数字。它有利于测试，不过有点毁游（san）戏（guan）的味道。这是最终源码：

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

完成！

这第一个项目展示了：`let`、`match`、方法、关联函数、使用外部包装箱等。

此刻，你成功地构建了猜猜看游戏！恭喜！

语法和语义

> syntax-and-semantics.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

这一部分将 Rust 折成小的部分，每一部分对应一个概念。

如果你想要自底向上的学习 Rust，按顺序阅读这一部分将会有很大帮助。

这些部分也组成了一个对各种概念的参考，所以如果你阅读其它教程并发现一些迷惑的问题，你可以在这里找到一些解释。

变量绑定

> variable-bindings.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

事实上每一个非“Hello World” Rust 程序都用了变量绑定。他们将一些值绑定到一个名字上，这样可以在之后使用他们。 `let` 被用来声明一个绑定，像这样：

```
fn main() {
    let x = 5;
}
```

在每个例子中都写上 `fn main() {` 有点冗长，所以之后我们将省略它。如果你是一路看过来的，确保你写了 `main()` 函数，而不是省略不写。否则，你将得到一个错误。

模式（Patterns）

在许多语言中，这叫做变量。不过 Rust 的变量绑定有一些不同的巧妙之处。例如 `let` 语句的左侧是一个“模式”，而不仅仅是一个变量。这意味着我们可以这样写：

```
let (x, y) = (1, 2);
```

在这个语句被计算后， `x` 将会是1，而 `y` 将会是2。模式非常强大，并且本书中有关于它的部分。我们现在还不需要这些功能，所以接下来你只需记住有这个东西就行了。

类型注解（Type annotations）

Rust 是一个静态类型语言，这意味着我们需要先确定我们需要的类型。那为什么我们第一个例子能编译过呢？好的，Rust有一个叫做类型推断的功能。如果它能确认这是什么类型，Rust 不需要你明确地指出来。

若你愿意，我们也可以加上类型。类型写在一个冒号（ `:` ）后面：

```
let x: i32 = 5;
```

如果我叫你对着全班同学大声读出这一行，你应该大喊“ `x` 被绑定为 `i32` 类型，它的值是 `5` ”。

在这个例子中我们选择 `x` 代表一个 32 位的有符号整数。Rust 有许多不同的原生整数类型。以 `i` 开头的代表有符号整数而 `u` 开头的代表无符号整数。可能的整数大小是 8、16、32 和 64 位。

在之后的例子中，我们可能会在注释中注明变量类型。例子看起来像这样：

```
fn main() {
    let x = 5; // x: i32
}
```

注意注释和 `let` 表达式有类似的语法。理想的 Rust 代码中不应包含这类注释。不过我们偶尔会这么做来帮助你理解 Rust 推断的是什么类型。

可变性（Mutability）

绑定默认是不可变的（*immutable*）。下面的代码将不能编译：

```
let x = 5;
x = 10;
```

它会给你如下错误：

```
error: re-assignment of immutable variable `x`
     x = 10;
     ^~~~~~~
```

如果你想一个绑定是可变的，使用 `mut` ：

```
let mut x = 5; // mut x: i32
x = 10;
```

不止一个理由使得绑定默认不可变的，不过我们可以通过一个 Rust 的主要目标来理解它：安全。如果你没有使用 `mut` ，编译器会捕获它，让你知道你改变了一个你可能并不打算让它改变的值。如果绑定默认是可变的，编译器将不可能告诉你这些。如果你确实想变量可变，解决办法也非常简单：加个 `mut` 。

尽量避免可变状态有一些其它好处，不过这不在这个教程的讨论范围内。大体上，你总是可以避免显式可变量，并且这也是 Rust 希望你做的。即便如此，有时，可变量是你需要的，所以这并不是被禁止的。

初始化绑定（Initializing bindings）

Rust 变量绑定有另一个不同于其它语言的方面：绑定要求在可以使用它之前必须初始化。

让我们尝试一下。将你的 `src/main.rs` 修改为为如下：

```
fn main() {
    let x: i32;

    println!("Hello world!");
}
```

你可以用 `cargo build` 命令去构建它。它依然会输出“Hello, world!”，不过你会得到一个警告：

```
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello
_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unus
ed_variables)] on by default
src/main.rs:2     let x: i32;
                      ^
```

Rust 警告我们从未使用过这个变量绑定，但是因为我们从未用过它，无害不罚。然而，如果你确实想使用 `x` ，事情就不一样了。让我们试一下。修改代码如下：

```
fn main() {
    let x: i32;

    println!("The value of x is: {}", x);
}
```

然后尝试构建它。你会得到一个错误：

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello
_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized vari
able: `x`
src/main.rs:4     println!("The value of x is: {}", x);
                                                    ^

note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.
```

Rust 是不会让我们使用一个没有经过初始化的值的。

让我们讨论一下我们添加到 `println!` 中的内容。

如果你输出的字符串中包含一对大括号（ `{}` ，一些人称之为胡须。。（译注：moustaches，八字胡））,Rust将把它解释为插入值的请求。字符串插值（*String interpolation*）是一个计算机科学术语，代表“在字符串中插入值”。我们加上一个逗号，然后是一个 `x` ，来表示我们想插入 `x` 的值。逗号用来分隔我们传递给函数和宏的参数，如果你想传递多个参数的话。

当你只写了大括号的时候，Rust 会尝试检查值的类型来显示一个有意义的值。如果你想指定详细的语法，有很多选项可供选择。现在，让我们保持默认格式，整数并不难打印。

作用域和隐藏（Scope and shadowing）

让我们回到绑定的话题上。变量绑定有一个作用域 - 他们被限制只能在他们被定义的块中存在。一个块是一个被 `{` 和 `}` 包围的语句集合。函数定义也是块！在下面的例子中我们定义了两个变量绑定， `x` 和 `y` ，他们位于不同的作用域中。 `x` 可以在 `fn main() {}` 块中被访问，而 `y` 只能在内部块内访问：

```
fn main() {
    let x: i32 = 17;
    {
        let y: i32 = 3;
        println!("The value of x is {} and value of y is {}", x,
 y);
    }
    println!("The value of x is {} and value of y is {}", x, y);
// This won't work.
}
```

第一个 `println!` 将会打印“The value of x is 17 and the value of y is 3”，不过这个并不能编译成功，因为第二个 `println!` 并不能访问 `y` 的值，因为它已不在作用域中。相反我们得到如下错误：

```
$ cargo build
   Compiling hello v0.1.0 (file:///home/you/projects/hello_world
)
main.rs:7:62: 7:63 error: unresolved name `y`. Did you mean `x`?
 [E0425]
main.rs:7     println!("The value of x is {} and value of y is {
}", x, y); // This won't work.

       ^
note: in expansion of format_args!
<std macros>:2:25: 2:56 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
main.rs:7:5: 7:65 note: expansion site
main.rs:7:62: 7:63 help: run `rustc --explain E0425` to see a de
tailed explanation
error: aborting due to previous error
Could not compile `hello`.

To learn more, run the command again with --verbose.
```

另外，变量可以被隐藏。这意味着一个后声明的并位于同一作用域的相同名字的变量绑定将会覆盖前一个变量绑定：

```
let x: i32 = 8;
{
    println!("{}", x); // Prints "8".
    let x = 12;
    println!("{}", x); // Prints "12".
}
println!("{}", x); // Prints "8".
let x =  42;
println!("{}", x); // Prints "42".
```

隐藏和可变绑定可能表现为同一枚硬币的两面，他们是两个不同的概念，不能互换使用。举个例子，隐藏允许我们将一个名字重绑定为不同的类型。它也可以改变一个绑定的可变性。注意隐藏并不改变和销毁被绑定的值，这个值会在离开作用域之

前继续存在，即便无法通过任何手段访问到它。

```
let mut x: i32 = 1;
x = 7;
let x = x; // `x` is now immutable and is bound to `7`

let y = 4;
let y = "I can also be bound to text!"; // `y` is now of a diffe
rent type
```

函数

> functions.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

到目前为止你应该见过一个函数，`main` 函数：

```
fn main() {
}
```

这可能是最简单的函数声明。就像我们之前提到的，`fn` 表示“这是一个函数”，后面跟着名字，一对括号因为这函数没有参数，然后是一对大括号代表函数体。下面是一个叫 `foo` 的函数：

```
fn foo() {
}
```

那么有参数是什么样的呢？下面这个函数打印一个数字：

```
fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

下面是一个使用了 `print_number` 函数的完整的程序：

```
fn main() {
    print_number(5);
}

fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

如你所见，函数参数与 `let` 声明非常相似：参数名加上冒号再加上参数类型。

下面是一个完整的程序，它将两个数相加并打印结果：

```
fn main() {
    print_sum(5, 6);
}

fn print_sum(x: i32, y: i32) {
    println!("sum is: {}", x + y);
}
```

在调用函数和声明函数时，你需要用逗号分隔多个参数。

与 `let` 不同，你必须为函数参数声明类型。下面代码将不能工作：

```
fn print_sum(x, y) {
    println!("sum is: {}", x + y);
}
```

你会获得如下错误：

```
expected one of `!`, `:`, or `@`, found `)`
fn print_sum(x, y) {
```

这是一个有意为之的设计决定。即使像 Haskell 这样的能够全程序推断的语言，注明类型也经常作为一个最佳实践而被建议。我们认为即使允许在在函数体中推断，也要强制函数声明参数类型。这是一个全推断与无推断的最佳平衡。

如果我们要一个返回值呢？下面这个函数给一个整数加一：

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

Rust 函数只能返回一个值，并且你需要在一个“箭头”后面声明类型，它是一个破折号（ `-` ）后跟一个大于号（ `>` ）。

注意这里并没有一个分号。如果你把它加上：

```
fn add_one(x: i32) -> i32 {
    x + 1;
}
```

你将会得到一个错误：

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
     x + 1;
}

help: consider removing this semicolon:
     x + 1;
          ^
```

这揭露了关于 Rust 两个有趣的地方：它是一个基于表达式的语言，并且分号与其它基于“大括号和分号”的语言不同。这两个方面是相关的。

表达式 VS 语句

Rust 主要是一个基于表达式的语言。只有两种语句，其它的一切都是表达式。

然而这又有什么区别呢？表达式返回一个值，而语句不是。这就是为什么这里我们以“不是所有控制路径都返回一个值”结束：`x + 1;` 语句不返回一个值。Rust 中有两种类型的语句：“声明语句”和“表达式语句”。其余的一切是表达式。让我们先讨论下声明语句。

在一些语言中，变量绑定可以被写成一个表达式，不仅仅是语句。例如 Ruby：

```
x = y = 5
```

然而，在 Rust 中，使用 `let` 引入一个绑定并不是一个表达式。下面的代码会产生一个编译时错误：

```
let x = (let y = 5); // expected identifier, found keyword `let`
```

编译器告诉我们这里它期望看到表达式的开头，而 `let` 只能开始一个语句，不是一个表达式。

注意赋值一个已经绑定过的变量（例如， `y = 5` ）仍是一个表达式，即使它的（返回）值并不是特别有用。不像其它语言中赋值语句返回它赋的值（例如，前面例子中的 `5` ），在 Rust 中赋值的值是一个空的元组 `()` ：

```
let mut y = 5;

let x = (y = 6);  // `x` has the value `()`, not `6`.
```

Rust中第二种语句是表达式语句。它的目的是把任何表达式变为语句。在实践环境中，Rust 语法期望语句后跟其它语句。这意味着你用分号来分隔各个表达式。这意味着Rust看起来很像大部分其它使用分号做为语句结尾的语言，并且你会看到分号出现在几乎每一行你看到的 Rust 代码。

那么我们说“几乎”的例外是神马呢？你已经见过它了，在这些代码中：

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

我们的函数声称它返回一个 `i32` ，但是带上个分号，它就会返回一个 `()` 。Rust 意识到这大概不是我们想要的，并在之前我们看到的错误中建议去掉分号。

提早返回（Early returns）

不过提早返回怎么破？Rust确实有这么一个关键字， `return` ：

```
fn foo(x: i32) -> i32 {
    return x;

    // we never run this code!
    x + 1
}
```

使用 `return` 作为函数的最后一行是可行的，不过被认为是一个糟糕的风格：

```
fn foo(x: i32) -> i32 {
    return x + 1;
}
```

如果你之前没有使用过基于表达式的语言，那么前面的没有 `return` 的定义可能看起来有点奇怪。不过它随着时间的推移它会变得直观。

发散函数（Diverging functions）

Rust有些特殊的语法叫“发散函数”，这些函数并不返回：

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

`panic!` 是一个宏，类似我们已经见过的 `println!()` 。与 `println!()` 不同的是， `panic!()` 导致当前的执行线程崩溃并返回指定的信息。因为这个函数会崩溃，所以它不会返回，所以它拥有一个类型 `!` ，它代表“发散”。

如果你添加一个叫做 `diverges()` 的函数并运行，你将会得到一些像这样的输出：

```
thread ‘main’ panicked at ‘This function never returns!’, hello.
rs:2
```

如果你想要更多信息，你可以设定 `RUST_BACKTRACE` 环境变量来获取 backtrace ：

```
$ RUST_BACKTRACE=1 ./diverges
thread 'main' panicked at 'This function never returns!', hello.
rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02
817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91
bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844
b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447
423903650
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479
104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

如果你需要覆盖一个已经设置的 `RUST_BACKTRACE` 的值，同时你又不能仅仅 unset 这个变量，这时把它设置成 `0` 来避免获得 backtrace。任何其它（非 0）值将打开 backtrace。

```
$ export RUST_BACKTRACE=1
...
$ RUST_BACKTRACE=0 ./diverges
thread '<main>' panicked at 'This function never returns!', hell
o.rs:2
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

`RUST_BACKTRACE` 也可以用于 Cargo 的 `run` 命令：

```
$ RUST_BACKTRACE=1 cargo run
     Running `target/debug/diverges`
thread '<main>' panicked at 'This function never returns!', hell
o.rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02
817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91
bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844
b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447
423903650
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479
104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

发散函数可以被用作任何类型：

```
# fn diverges() -> ! {
#    panic!("This function never returns!");
# }
let x: i32 = diverges();
let x: String = diverges();
```

函数指针

我们也可以创建指向函数的变量绑定：

```
let f: fn(i32) -> i32;
```

`f` 是一个指向一个获取 `i32` 作为参数并返回 `i32` 的函数的变量绑定。例如：

```
fn plus_one(i: i32) -> i32 {
    i + 1
}

// without type inference
let f: fn(i32) -> i32 = plus_one;

// with type inference
let f = plus_one;
```

你可以用 `f` 来调用这个函数：

```
# fn plus_one(i: i32) -> i32 { i + 1 }
# let f = plus_one;
let six = f(5);
```

原生类型

> primitive-types.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

Rust 有一系列被认为是“原生”的类型。这意味着它们是内建在语言中的。Rust被构建为在标准库中也提供了一些建立在这些类型之上的有用的类型，不过它们也大部分是原生的。

布尔型

Rust 有一个内建的布尔类型，叫做 `bool` 。它有两个值， `true` 和 `false` ：

```
let x = true;

let y: bool = false;
```

布尔型通常用在if语句中。

你可以在标准库文档中找到更多关于 `bool` 的文档。

`char`

`char` 类型代表一个单独的 Unicode 字符的值。你可以用单引号（ `'` ）创建 `char` ：

```
let x = 'x';
let two_hearts = '';
```

不像其它语言，这意味着Rust的 `char` 并不是 1 个字节，而是 4 个。

你可以在标准库文档中找到更多关于 `char` 的文档。

数字类型

Rust有一些分类的大量数字类型：有符号和无符号，定长和变长，浮点和整型。

这些类型包含两部分：分类，和大小。例如， `u16` 是一个拥有16位大小的无符号类型。更多字节让你拥有更大的数字。

如果一个数字常量没有推断它类型的条件，它采用默认类型：

```
let x = 42; // `x` has type `i32`.

let y = 1.0; // `y` has type `f64`.
```

这里有一个不同数字类型的列表，以及它们在标准库中的文档：

- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- isize
- usize
- f32
- f64

让我们按分类重温一遍：

有符号和无符号

整型有两种变体：有符号和无符号。为了理解它们的区别，让我们考虑一个4比特大小的数字。一个有符号，4比特数字你可以储存 `-8` 到 `+7` 的数字。有符号数采用“二进制补码”表示。一个无符号4比特的数字，因为它不需要储存负数，可以出储存 `0` 到 `+15` 的数字。

无符号类型使用 `u` 作为他们的分类，而有符号类型使用 `i` 。 `i` 代表“integer”。所以 `u8` 是一个8位无符号数字，而 `i8` 是一个8位有符号数字。

固定大小类型

固定大小类型在其表现中有特定数量的位。有效的位大小是 `8` ，`16` ，`32` 和 `64` 。那么，`u32` 是无符号的，32 位整型，而 `i64` 是有符号，64 位整型。

可变大小类型

Rust 也提供了依赖底层机器指针大小的类型。这些类型拥有“size”分类，并有有符号和无符号变体。它有两个类型：`isize` 和 `usize` 。

浮点类型

Rust 也有两个浮点类型：`f32` 和 `f64` 。它们对应 IEEE-754 单精度和双精度浮点数。

数组

像很多编程语言一样，Rust有用来表示数据序列的列表类型。最基本的是数组，一个定长相同类型的元素列表。数组默认是不可变的。

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

数组的类型是 `[T; N]` 。我们会在泛型部分的时候讨论这个 `T` 标记。`N` 是一个编译时常量，代表数组的长度。

有一个可以将数组中每一个元素初始化为相同值的简写。在这个例子中，`a` 的每个元素都被初始化为 `0` ：

```
let a = [0; 20]; // a: [i32; 20]
```

你可以用 `a.len()` 来获取数组 `a` 的元素数量：

```
let a = [1, 2, 3];

println!("a has {} elements", a.len());
```

你可以用下标（*subscript notation*）来访问特定的元素：

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

就跟大部分编程语言一个样，下标从0开始，所以第一个元素是 `names[0]` ，第二个是 `names[1]` 。上面的例子会打印出 `The second name is: Brian` 。如果你尝试使用一个不在数组中的下标，你会得到一个错误：数组访问会在运行时进行边界检查。这种不适当的访问是其它系统编程语言中很多bug的根源。

你可以在标准库文档中找到更多关于 `array` 的文档。

切片（Slices）

一个切片（*slice*）是一个数组的引用（或者"视图"）。它有利于安全，有效的访问数组的一部分而不用进行拷贝。比如，你可能只想要引用读入到内存的文件中的一行。原理上，片段并不是直接创建的，而是引用一个已经存在的变量。片段有预定义的长度，可以是可变也可以是不可变的。

在底层，slice 代表一个指向数据开始的指针和一个长度。

切片语法（Slicing syntax）

你可以用一个 `&` 和 `[]` 的组合从多种数据类型创建一个切片。 `&` 表明切片类似于引用，这个我们会在本部分的后面详细介绍。带有一个范围的 `[]` ，允许你定义切片的长度：

```
let a = [0, 1, 2, 3, 4];
let complete = &a[..]; // A slice containing all of the elements
 in `a`.
let middle = &a[1..4]; // A slice of `a`: only the elements `1`,
 `2`, and `3`.
```

片段拥有 `&[T]` 类型。当我们涉及到泛型时会讨论这个 `T` 。

你可以在标准库文档中找到更多关于 `slices` 的文档。

`str`

Rust的 `str` 类型是最原始的字符串类型。作为一个不定长类型，它本身并不是非常有用，不过当它用在引用后是就有用了，例如&str。如你所见，我们到时候再讲。

你可以在标准库文档中找到更多关于 `str` 的文档。

元组（Tuples）

元组（tuples）是固定大小的有序列表。如下：

```
let x = (1, "hello");
```

这是一个长度为 2 的元组，由括号和逗号组成。下面也是同样的元组，不过注明了数据类型：

```
let x: (i32, &str) = (1, "hello");
```

如你所见，元组的类型跟元组看起来很像，只不过类型取代的值的位置。细心的读者可能会注意到元组是异质的：这个元组中有一个 `i32` 和一个 `&str` 。在系统编程语言中，字符串要比其它语言中来的复杂。现在，可以认为 `&str` 是一个字符串片段（*string slice*），我们马上会讲到它。

你可以把一个元组赋值给另一个，如果它们包含相同的类型和数量。当元组有相同的长度时它们有相同的数量。

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

你可以通过一个解构**let**（*destructuring let*）访问元组中的字段。下面是一个例子：

```
let (x, y, z) = (1, 2, 3);

println!("x is {}", x);
```

还记得之前我曾经说过 `let` 语句的左侧远比一个赋值绑定强大吗？这就是证据。我们可以在 `let` 左侧写一个模式，如果它能匹配右侧的话，我们可以一次写多个绑定。这种情况下， `let` “解构”或“拆开”了元组，并分成了三个绑定。

这个模式是很强大的，我们后面会经常看到它。

你可以一个逗号来消除一个单元素元组和一个括号中的值的歧义：

```
(0,); // single-element tuple
(0); // zero in parentheses
```

元组索引（**Tuple Indexing**）

你也可以用索引语法访问一个元组的字段：

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

println!("x is {}", x);
```

就像数组索引，它从 `0` 开始，不过也不像数组索引，它使用 `.` ，而不是 `[]` 。

你可以在标准库文档中找到更多关于 `tuple` 的文档。

函数

函数也有一个类型！它们看起来像这样：

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

在这个例子中， `x` 是一个“函数指针”，指向一个获取一个 `i32` 参数并返回一个 `i32` 值的函数。

注释

> comments.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

现在我们写了一些函数，是时候学习一下注释了。注释是你帮助其他程序员理解你的代码的备注。编译器基本上会忽略它们。

Rust 有两种需要你了解的注释格式：行注释（*line comments*）和文档注释（*doc comments*）。

```
// Line comments are anything after ‘//’ and extend to the end of the line.

let x = 5; // This is also a line comment.

// If you have a long explanation for something, you can put line comments next
// to each other. Put a space between the // and your comment so that it’s
// more readable.
```

另一种注释是文档注释。文档注释使用 `///` 而不是 `//`，并且内建 Markdown 标记支持：

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// # fn add_one(x: i32) -> i32 {
/// #     x + 1
/// # }
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

有另外一种风格的文档注释，`//!`，用来注释包含它的项（也就是说，crate，模块或者函数），而不是位于它之后的项。它经常用在 crate 根文件（lib.rs）或者模块根文件（mod.rs）：

```
//! # The Rust Standard Library
//!
//! The Rust Standard Library provides the essential runtime
//! functionality for building portable Rust software.
```

当书写文档注释时，加上参数和返回值部分并提供一些用例将是非常，非常有帮助的。你会注意到我们在这里用了一个新的宏：`assert_eq!`。它比较两个值，并当它们不相等时`panic!`。这在文档中是非常有帮助的。还有一个宏，`assert!`，它在传递给它的值是`false`的时候`panic!`。

你可以使用rustdoc工具来将文档注释生成为 HTML 文档，也可以将代码示例作为测试运行！

If

> if.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

Rust 的 `if` 并不是特别复杂，不过你会发现它更像动态类型语言而不是更传统的系统语言。所以让我来说说它，以便你能把握这些细节。

`if` 语句是分支这个更加宽泛的概念的一个特定形式。它的名字来源于树的树枝：一个选择点，根据选择的不同，将会使用不同的路径。

在 `if` 语句中，有一个引向两条路径的选择：

```
let x = 5;

if x == 5 {
    println!("x is five!");
}
```

如果在什么别的地方更改了 `x` 的值，这一行将不会输出。更具体一点，如果 `if` 后面的表达式的值为 `true` ，这个代码块将被执行。为 `false` 则不被执行。

如果你想当值为 `false` 时执行些什么，使用 `else` ：

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else {
    println!("x is not five :(");
}
```

如果不止一种情况，使用 `else if` ：

```rust
let x = 5;

if x == 5 {
    println!("x is five!");
} else if x == 6 {
    println!("x is six!");
} else {
    println!("x is not five or six :(");
}
```

这些都是非常标准的情况。然而你也可以这么写：

```rust
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

你可以（或许也应该）这么写：

```rust
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

这代码可以被执行是因为 `if` 是一个表达式。表达式的值是任何被选择的分支的最后一个表达式的值。一个没有 `else` 的 `if` 总是返回 `()` 作为返回值。

循环

> loops.md
> commit 549270d4e66cc77b1f02cea13aaad9c90f150524

Rust 目前提供 3 种方法来进行一些迭代操作。他们是 `loop` ， `while` 和 `for` 。每种方法都有自己的用途。

loop

无限 `loop` 是 Rust 提供的最简单的循环。使用 `loop` 关键字，Rust 提供了一个直到一些终止语句被执行的循环方法。Rust 的无限 `loop` 看起来像这样：

```
loop {
    println!("Loop forever!");
}
```

while

Rust 也有一个 `while` 循环。它看起来像：

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

`while` 循环是当你不确定应该循环多少次时正确的选择。

如果你需要一个无限循环，你可能想要这么写：

```
while true {
```

然而， `loop` 远比它适合处理这个情况：

```
loop {
```

Rust 的控制流分析会区别对待这个与 `while true` ，因为我们知道它会一直循环。现阶段理解这些细节意味着什么并不是非常重要，基本上，你给编译器越多的信息，越能确保安全和生成更好的代码，所以当你打算无限循环的时候应该总是倾向于使用 `loop` 。

for

`for` 用来循环一个特定的次数。然而，Rust的 `for` 循环与其它系统语言有些许不同。Rust的 `for` 循环看起来并不像这个“C语言样式”的 `for` 循环：

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

相反，它看起来像这个样子：

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

更抽象的形式：

```
for var in expression {
    code
}
```

这个表达式是一个迭代器.迭代器返回一系列的元素。每次迭代循环中的一个元素。然后它的值与 `var` 绑定，它在循环体中有效。每当循环体执行完后，我们从迭代器中取出下一个值，然后我们再重复一遍。当迭代器中不再有值时，`for` 循环结束。

在我们的例子中，`0..10` 表达式取一个开始和结束的位置，然后给出一个含有这之间值得迭代器。当然它不包括上限值，所以我们的循环会打印 `0` 到 `9`，而不是到 `10`。

Rust 没有使用“C语言风格”的 `for` 循环是有意为之的。即使对于有经验的 C 语言开发者来说，要手动控制要循环的每个元素也都是复杂并且易于出错的。

Enumerate 方法

当你需要记录你已经循环了多少次了的时候，你可以使用 `.enumerate()` 函数。

对于范围（On ranges）：

```
for (index, value) in (5..10).enumerate() {
    println!("index = {} and value = {}", index, value);
}
```

输出：

```
index = 0 and value = 5
index = 1 and value = 6
index = 2 and value = 7
index = 3 and value = 8
index = 4 and value = 9
```

别忘了在范围外面加上括号。

对于迭代器（On iterators）：

```
let lines = "hello\nworld".lines();

for (linenumber, line) in lines.enumerate() {
    println!("{}: {}", linenumber, line);
}
```

输出：

```
0: hello
1: world
```

提早结束迭代（Ending iteration early）

让我们再看一眼之前的`while`循环：

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

我们必须使用一个`mut`布尔型变量绑定，`done`,来确定何时我们应该推出循环。Rust 有两个关键字帮助我们来修改迭代：`break`和`continue`。

这样，我们可以用`break`来写一个更好的循环：

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

现在我们用 `loop` 来无限循环，然后用 `break` 来提前退出循环。

`continue` 比较类似，不过不是退出循环，它直接进行下一次迭代。下面的例子只会打印奇数：

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

    println!("{}", x);
}
```

`break` 和 `continue` 在 `while` 循环和 [`for` 循环]中都有效。

循环标签（Loop labels）

你也许会遇到这样的情形，当你有嵌套的循环而希望指定你的哪一个 `break` 或 `continue` 该起作用。就像大多数语言，默认 `break` 或 `continue` 将会作用于最内层的循环。当你想要一个 `break` 或 `continue` 作用于一个外层循环，你可以使用标签来指定你的 `break` 或 `continue` 语句作用的循环。如下代码只会在 `x` 和 `y` 都为奇数时打印他们：

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // Continues the loop
 over `x`.
        if y % 2 == 0 { continue 'inner; } // Continues the loop
 over `y`.
        println!("x: {}, y: {}", x, y);
    }
}
```

Vectors

> vectors.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

“Vector”是一个动态或“可增长”的数组，被实现为标准库类型 `Vec<T>` （其中 `<T>` 是一个泛型语句）。vector 总是在堆上分配数据。vector 与切片就像 `String` 与 `&str` 一样。你可以使用 `vec!` 宏来创建它：

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
```

（与之前使用 `println!` 宏时不一样，我们用中括号 `[]` 配合 `vec!` 。为了方便，Rust 允许使用上述各种情况。）

对于重复初始值有另一种形式的 `vec!` ：

```
let v = vec![0; 10]; // ten zeroes
```

vector 将它们的内容以连续的 `T` 的数组的形式存储在堆上，这意味着它们必须在编译时就知道 `T` 的大小（就是存储一个 `T` 需要多少字节）。有些类型的大小不可能在编译时就知道。为此你需要保存一个指向该类型的指针：幸好， `Box` 类型正好适合这种情况。

访问元素

为了vector特定索引的值，使用 `[]` ：

```
let v = vec![1, 2, 3, 4, 5];

println!("The third element of v is {}", v[2]);
```

索引从 `0` 开始，所以第 3 个元素是 `v[2]` 。

另外值得注意的是必须用 `usize` 类型的值来索引：

```
let v = vec![1, 2, 3, 4, 5];

let i: usize = 0;
let j: i32 = 0;

// Works:
v[i];

// Doesn’t:
v[j];
```

用非 `usize` 类型索引的话会给出类似如下的错误：

```
error: the trait bound `collections::vec::Vec<_> : core::ops::In
dex<i32>`
is not satisfied [E0277]
v[j];
^~~~
note: the type `collections::vec::Vec<_>` cannot be indexed by `
i32`
error: aborting due to previous error
```

信息中有很多标点符号，不过关键是：你不能用 `i32` 来索引。

越界访问（Out-of-bounds Access）

如果尝试访问并不存在的索引：

```
let v = vec![1, 2, 3];
println!("Item 7 is {}", v[7]);
```

那么当前的线程会 panic并输出如下信息：

```
thread 'main' panicked at 'index out of bounds: the len is 3 but
 the index is 7'
```

如果你想处理越界错误而不是 panic，你可以使用像 `get` 或 `get_mut` 这样的方法，他们当给出一个无效的索引时返回 `None` ：

```
let v = vec![1, 2, 3];
match v.get(7) {
    Some(x) => println!("Item 7 is {}", x),
    None => println!("Sorry, this vector is too short.")
}
```

迭代

可以用 `for` 来迭代 vector 的元素。有3个版本：

```
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("A reference to {}", i);
}

for i in &mut v {
    println!("A mutable reference to {}", i);
}

for i in v {
    println!("Take ownership of the vector and its element {}",
i);
}
```

注意：你不能在使用 vector 的所有权遍历之后再次遍历它。你可以使用它的引用多次遍历 vector。例如，下面的代码不能编译。

```
let v = vec![1, 2, 3, 4, 5];

for i in v {
    println!("Take ownership of the vector and its element {}",
i);
}

for i in v {
    println!("Take ownership of the vector and its element {}",
i);
}
```

而如下代码则可以完美运行：

```
let v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("This is a reference to {}", i);
}

for i in &v {
    println!("This is a reference to {}", i);
}
```

vector还有很多有用的方法，可以看看vector的API文档了解它们。

所有权

> ownership.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

这篇教程是现行 3 个 Rust 所有权系统章节的第一部分。所有权系统是Rust最独特且最引人入胜的特性之一，也是作为 Rust 开发者应该熟悉的。Rust 所追求最大的目标 -- 内存安全，关键在于所有权。所有权系统有一些不同的概念，每个概念独自成章：

- 所有权，你正在阅读的这个章节
- 借用，以及它关联的特性: "引用" (references)
- 生命周期，关于借用的高级概念

这3章依次互相关联，你需要完整地阅读全部 3 章来对Rust的所有权系统进行全面的了解。

原则（Meta）

在我们开始详细讲解之前，这有两点关于所有权系统重要的注意事项。

Rust 注重安全和速度。它通过很多零开销抽象（*zero-cost abstractions*）来实现这些目标，也就是说在 Rust 中，实现抽象的开销尽可能的小。所有权系统是一个典型的零开销抽象的例子。本文提到所有的分析都是在编译时完成的。你不需要在运行时为这些功能付出任何开销。

然而，这个系统确实有一个开销：学习曲线。很多 Rust 初学者会经历我们所谓的“与借用检查器作斗争”的过程，也就是指 Rust 编译器拒绝编译一个作者认为合理的程序。这种“斗争”会因为程序员关于所有权系统如何工作的基本模型与 Rust 实现的实际规则不匹配而经常发生。当你刚开始尝试 Rust 的时候，你很可能会有相似的经历。然而有一个好消息：更有经验的 Rust 开发者反映，一旦他们适应所有权系统一段时间之后，与借用检查器的冲突会越来越少。

记住这些之后，让我们来学习关于所有权的内容。

所有权（Ownership）

Rust 中的变量绑定有一个属性：它们有它们所绑定的的值的所有权。这意味着当一个绑定离开作用域，它们绑定的资源就会被释放。例如：

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

当 `v` 进入作用域，一个新的 vector 在栈上被创建，并在堆上为它的3个元素分配了空间。当 `v` 在 `foo()` 的末尾离开作用域，Rust将会清理掉与向量（vector）相关的一切，甚至是堆上分配的内存。这在作用域的结尾是一定（deterministically）会发生的。

我们在之前的章节介绍了vector；这里我们只是用它来作为一个在运行时在堆上分配内存的类型的例子。他们表现起来像数组，除了通过 `push()` 更多元素他们的大小会改变。

vector 有一个泛型类型 `Vec<T>` ，所以在这个例子中 `v` 是 `Vec<i32>` 类型的。我们将会在本章的后面详细介绍泛型。

移动语义（Move semantics）

然而这里有更巧妙的地方：Rust 确保了对于任何给定的资源都正好（只）有一个绑定与之对应。例如，如果我们有一个 vector，我们可以把它赋予另外一个绑定：

```
let v = vec![1, 2, 3];

let v2 = v;
```

不过，如果之后我们尝试使用 `v` ，我们得到一个错误：

```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] is: {}", v[0]);
```

它看起来像这样：

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

当我们定义了一个取得所有权的函数，并尝试在我们把变量作为参数传递给函数之后使用这个变量时，会发生相似的事情：

```
fn take(v: Vec<i32>) {
    // What happens here isn’t important.
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] is: {}", v[0]);
```

一样的错误："use of moved value"。当我们把所有权转移给别的绑定时，我们说我们"移动"了我们引用的值。这里你并不需要什么类型的特殊注解，这是 Rust 的默认行为。

细节

在移动了绑定后我们不能使用它的原因是微妙的，也是重要的。当我们写了这样的代码：

```
let x = 10;
```

Rust 在栈上为一个整型（ `i32` ）分配了内存，将代表值 10 的位拷贝到分配的内存中并将这个内存区域绑定到变量 x 上以供进一步的引用。

现在考虑一下下面的代码段：

```
let v = vec![1, 2, 3];

let mut v2 = v;
```

第一行在栈上为 vector 对象 `v` 分配了内存，就像上面的 x 一样。不过同时它也在堆上为实际的数据 [1, 2, 3] 分配了一些内存。Rust 拷贝堆上分配的内存的地址到一个内部指针，作为位于栈上的 vector 对象的一部分（让我们叫它数据指针）。值得指出的是（即便冒着冗余的风险），我们也将 vector 对象和它的数据存储在不同的内存区域，而不是分配到一块连续的内存中（出于一些我们现在不会讨论的理由）。vector 的这两部分（在堆上的和在栈上的）在任何时候都必须同步像大小，容量这样的信息。

当我们从 `v` 移动到 `v2` ，Rust 实际上按位拷贝了 `v` 在栈上分配的内容到 `v2` 。这个浅拷贝（shallow copy）并没有复制一份堆上分配的实际数据。这就是说将会有两个 vector 内容的指针都指向通用的堆上分配的空间。这样会违反 Rust 的安全保证，通过引入一个数据竞争，当你可以同时访问 `v` 和 `v2` 的时候。

例如如果我们通过 `v2` 缩短 vector 到只有两个元素：

```
# let v = vec![1, 2, 3];
# let mut v2 = v;
v2.truncate(2);
```

而同时 `v` 仍是可以访问的话，这会产生一个无效的 vector，因为它并不知道堆上的数据已经被缩短了。现在 `v` 在栈上的部分与堆上的相应部分的信息并不一致。 `v` 仍然认为有 3 个元素并乐意我们访问那个并不存在的元素 `v[2]` ，不过你可能已经知道这是一个导致灾难的剧本。因为这可能会导致一个段错误，更糟的是会允许未经授权的用户读取他没有访问权限的数据。

这就是为何 Rust 在我们移动后禁止使用 `v` 的原因。

注意到优化可能会根据情况移除栈上字节（例如上面的向量）的实际拷贝也是很重要的。所以它也许并不像它开始看起来那样没有效率。

`Copy` 类型

我们已经知道了当所有权被转移给另一个绑定以后，你不能再使用原始绑定。然而，有一个trait会改变这个行为，它叫做 `Copy` 。我们还没有讨论到 trait，不过目前，你可以理解为一个为特定类型增加额外行为的标记。例如：

```
let v = 1;

let v2 = v;

println!("v is: {}", v);
```

在这个情况， `v` 是一个 `i32` ，它实现了 `Copy` 。这意味着，就像一个移动，当我们把 `v` 赋值给 `v2` ,产生了一个数据的拷贝。不过，不像一个移动，我们仍可以在之后使用 `v` 。这是因为 `i32` 并没有指向其它数据的指针，对它的拷贝是一个完整的拷贝。

所有基本类型都实现了 `Copy` trait，因此他们的所有权并不像你想象的那样遵循“所有权规则”被移动。作为一个例子，如下两段代码能够编译是因为 `i32` 和 `bool` 类型实现了 `Copy` trait。

```
fn main() {
    let a = 5;

    let _y = double(a);
    println!("{}", a);
}

fn double(x: i32) -> i32 {
    x * 2
}
```

```
fn main() {
    let a = true;

    let _y = change_truth(a);
    println!("{}", a);
}

fn change_truth(x: bool) -> bool {
    !x
}
```

如果我们使用了没有实现`Copy` trait的类型，我们会得到一个编译错误，因为我们尝试使用一个移动了的值。

```
error: use of moved value: `a`
println!("{}", a);
               ^
```

我们会在trait部分讨论如何编写你自己类型的`Copy`。

所有权之外（More than ownership）

当然，如果我们不得不在每个我们写的函数中交还所有权：

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // Do stuff with `v`.

    // // Hand back ownership.
    v
}
```

这将会变得烦人。它在我们获取更多变量的所有权时变得更糟：

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32)
{
    // Do stuff with `v1` and `v2`.

    // Hand back ownership, and the result of our function.
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

额！返回值，返回的代码行（上面的最后一行），和函数调用都变得更复杂了。

幸运的是，Rust 提供了一个 trait，借用，它帮助我们解决这个问题。这个主题将在下一个部分讨论！

引用和借用

> references-and-borrowing.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

这篇教程是现行 3 个 Rust 所有权系统章节的第二部分。所有权系统是 Rust 最独特且最引人入胜的特性之一，也是作为 Rust 开发者应该熟悉的。Rust 所追求最大的目标 -- 内存安全，关键在于所有权。所有权系统有一些不同的概念，每个概念独自成章：

- 所有权，关键章节
- 借用，你正在阅读的这个章节
- 生命周期，关于借用的高级概念

这 3 章依次互相关联，你需要完整地阅读全部 3 章来对 Rust 的所有权系统进行全面的了解。

原则（Meta）

在我们开始详细讲解之前，这有两点关于所有权系统重要的注意事项。

Rust 注重安全和速度。它通过很多零开销抽象（*zero-cost abstractions*）来实现这些目标，也就是说在 Rust 中，实现抽象的开销尽可能的小。所有权系统是一个典型的零开销抽象的例子。本文提到所有的分析都是在编译时完成的。你不需要在运行时为这些功能付出任何开销。

然而，这个系统确实有一个开销：学习曲线。很多 Rust 初学者会经历我们所谓的“与借用检查器作斗争”的过程，也就是指 Rust 编译器拒绝编译一个作者认为合理的程序。这种“斗争”会因为程序员关于所有权系统如何工作的基本模型与 Rust 实现的实际规则不匹配而经常发生。当你刚开始尝试 Rust 的时候，你很可能会有相似的经历。然而有一个好消息：更有经验的 Rust 开发者反映，一旦他们适应所有权系统一段时间之后，与借用检查器的冲突会越来越少。

记住这些之后，让我们来学习关于借用的内容。

借用

在所有权章节的最后，我们有一个看起来像这样的糟糕的函数：

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32)
{
    // Do stuff with `v1` and `v2`.

    // Hand back ownership, and the result of our function.
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

然而这并不是理想的 Rust 代码，因为它没有利用'借用'这个编程语言的特点。这是它的第一步：

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // Do stuff with `v1` and `v2`.

    //  Return the answer.
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// We can use `v1` and `v2` here!
```

一个更具体的例子：

```
fn main() {
    // Don't worry if you don't understand how `fold` works, the
 point here is that an immutable reference is borrowed.
    fn sum_vec(v: &Vec<i32>) -> i32 {
        return v.iter().fold(0, |a, &b| a + b);
    }
    // Borrow two vectors and sum them.
    // This kind of borrowing does not allow mutation through th
e borrowed reference.
    fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
        // Do stuff with `v1` and `v2`.
        let s1 = sum_vec(v1);
        let s2 = sum_vec(v2);
        // Return the answer.
        s1 + s2
    }

    let v1 = vec![1, 2, 3];
    let v2 = vec![4, 5, 6];

    let answer = foo(&v1, &v2);
    println!("{}", answer);
}
```

与其获取 `Vec<i32>` 作为我们的参数，我们获取一个引用： `&Vec<i32>` 。并与其直接传递 `v1` 和 `v2` ，我们传递 `&v1` 和 `&v2` 。我们称 `&T` 类型为一个"引用"，而与其拥有这个资源，它借用了所有权。一个借用变量的绑定在它离开作用域时并不释放资源。这意味着 `foo()` 调用之后，我们可以再次使用原始的绑定。

引用是不可变的，就像绑定一样。这意味着在 `foo()` 中，向量完全不能被改变：

```
fn foo(v: &Vec<i32>) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

有如下错误：

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

放入一个值改变了向量，所以我们不允许这样做

`&mut` 引用

这有第二种类型的引用： `&mut T` 。一个“可变引用”允许你改变你借用的资源。例如：

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

这会打印 `6` 。我们让 `y` 是一个 `x` 的可变引用，接着把 `y` 指向的值加一。你会注意到 `x` 也必须被标记为 `mut` ，如果它不是，我们不能获取一个不可变值的可变引用。

你也会发现我们在 `y` 前面加了一个星号（ `*` ），成了 `*y` ，这是因为 `y` 是一个 `&mut` 引用。你也需要使用他们（星号）来访问引用的内容。

否则， `&mut` 引用就像一个普通引用。这两者之间,以及它们是如何交互的有巨大的区别。你会发现在上面的例子有些不太靠谱，因为我们需要额外的作用域，包围在 `{` 和 `}` 之间。如果我们移除它们，我们得到一个错误：

```
error: cannot borrow `x` as immutable because it is also borrowe
d as mutable
    println!("{}", x);
                   ^
note: previous borrow of `x` occurs here; the mutable borrow pre
vents
subsequent moves, borrows, or modification of `x` until the borr
ow ends
        let y = &mut x;
                     ^
note: previous borrow ends here
fn main() {

}
^
```

正如这个例子表现的那样，有一些规则是你必须要掌握的。

规则

Rust 中的借用有一些规则：

第一，任何借用必须位于比拥有者更小的作用域。第二，对于同一个资源（resource）的借用，以下情况不能同时出现在同一个作用域下：

- 1 个或多个不可变引用（`&T`）
- 唯一 1 个可变引用（`&mut T`）

`译者注：即同一个作用域下，要么只有一个对资源 A 的可变引用（&mut T），要么有 N 个不可变引用（&T），但不能同时存在可变和不可变的引用`

你可能注意到这些看起来很眼熟，虽然并不完全一样，它类似于数据竞争的定义：

> 当 2 个或更多个指针同时访问同一内存位置，当它们中至少有 1 个在写，同时操作并不是同步的时候存在一个“数据竞争”

通过引用，你可以拥有你想拥有的任意多的引用，因为它们没有一个在写。如果你在写，并且你需要2个或更多相同内存的指针，则你只能一次拥有一个 `&mut` 。这就是Rust如何在编译时避免数据竞争：如果打破规则的话，我们会得到错误。

在记住这些之后，让我们再次考虑我们的例子。

理解作用域（Thinking in scopes）

这是代码：

```
fn main() {
    let mut x = 5;
    let y = &mut x;

    *y += 1;

    println!("{}", x);
}
```

这些代码给我们如下错误：

```
error: cannot borrow `x` as immutable because it is also borrowe
d as mutable
    println!("{}", x);
                   ^
```

这是因为我们违反了规则：我们有一个指向 `x` 的 `&mut T` ，所以我们不允许创建任何 `&T` 。一个或另一个。错误记录提示了我们应该如何理解这个错误：

```
note: previous borrow ends here
fn main() {

}
^
```

换句话说，可变借用在剩下的例子中一直存在。我们需要的是可变借用在我们尝试调用 `println!` 之前结束并生成一个不可变借用。在 Rust 中，借用绑定在借用有效的作用域上。而我们的作用域看起来像这样：

```
fn main() {
    let mut x = 5;

    let y = &mut x;     // -+ &mut borrow of `x` starts here.
                        //  |
    *y += 1;            //  |
                        //  |
    println!("{}", x); // -+ - Try to borrow `x` here.
}                       // -+ &mut borrow of `x` ends here.
```

这些作用域冲突了：我们不能在 `y` 在作用域中时生成一个 `&x` 。

所以我们增加了一个大括号：

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut borrow starts here.
    *y += 1;        //  |
}                   // -+ ... and ends here.

println!("{}", x);  // <- Try to borrow `x` here.
```

这就没有问题了。我们的可变借用在我们创建一个不可变引用之前离开了作用域。不过作用域是看清一个借用持续多久的关键。

借用避免的问题（**Issues borrowing prevents**）

为什么要有这些限制性规则？好吧，正如我们记录的，这些规则避免了数据竞争。数据竞争能造成何种问题呢？这里有一些。

迭代器失效（**Iterator invalidation**）

一个例子是“迭代器失效”，它在当你尝试改变你正在迭代的集合时发生。Rust 的借用检查器阻止了这些发生：

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
}
```

这会打印出 1 到 3。因为我们在向量上迭代，我们只得到了元素的引用。同时 `v` 本身作为不可变借用，它意味着我们在迭代时不能改变它：

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

这里是错误：

```
error: cannot borrow `v` as mutable because it is also borrowed
as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable borrow p
revents
subsequent moves or mutable borrows of `v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!("{}", i);
    v.push(34);
}
^
```

我们不能修改 `v` 因为它被循环借用。

释放后使用

引用必须与它引用的值存活得一样长。Rust 会检查你的引用的作用域来保证这是正确的。

如果 Rust 并没有检查这个属性，我们可能意外的使用了一个无效的引用。例如：

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

println!("{}", y);
```

我们得到这个错误：

```
error: `x` does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following sta
tement 0 at
2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix f
ollowing
statement 0 at 4:18
    let x = 5;
    y = &x;
}
```

换句话说，`y` 只在 `x` 存在的作用域中有效。一旦 `x` 消失，它变成无效的引用。为此，这个错误说借用“并没有存活得足够久”因为它在应该有效的时候是无效的。

当引用在它引用的变量之前声明会导致类似的问题。这是因为同一作用域中的资源以他们声明相反的顺序被释放：

```
let y: &i32;
let x = 5;
y = &x;

println!("{}", y);
```

我们得到这个错误：

```
error: `x` does not live long enough
y = &x;
     ^
note: reference must be valid for the block suffix following sta
tement 0 at
2:16...
    let y: &i32;
    let x = 5;
    y = &x;

    println!("{}", y);
}

note: ...but borrowed value is only valid for the block suffix f
ollowing
statement 1 at 3:14
    let x = 5;
    y = &x;

    println!("{}", y);
}
```

在上面的例子中，`y` 在 `x` 之前被声明，意味着 `y` 比 `x` 生命周期更长，这是不允许的。

生命周期

> lifetimes.md
>
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

这篇教程是现行 3 个 Rust 所有权系统章节的第三部分。所有权系统是 Rust 最独特且最引人入胜的特性之一，也是作为 Rust 开发者应该熟悉的。Rust 所追求最大的目标 -- 内存安全，关键在于所有权。所有权系统有一些不同的概念，每个概念独自成章：

- 所有权，关键章节
- 借用，以及它关联的特性: "引用" (references)
- 生命周期，你正在阅读的这个章节

这 3 章依次互相关联，你需要完整地阅读全部 3 章来对 Rust 的所有权系统进行全面的了解。

原则（**Meta**）

在我们开始详细讲解之前，这有两点关于所有权系统重要的注意事项。

Rust 注重安全和速度。它通过很多零开销抽象（*zero-cost abstractions*）来实现这些目标，也就是说在 Rust 中，实现抽象的开销尽可能的小。所有权系统是一个典型的零开销抽象的例子。本文提到所有的分析都是在编译时完成的。你不需要在运行时为这些功能付出任何开销。

然而，这个系统确实有一个开销：学习曲线。很多 Rust 初学者会经历我们所谓的“与借用检查器作斗争”的过程，也就是指 Rust 编译器拒绝编译一个作者认为合理的程序。这种“斗争”会因为程序员关于所有权系统如何工作的基本模型与 Rust 实现的实际规则不匹配而经常发生。当你刚开始尝试 Rust 的时候，你很可能会有相似的经历。然而有一个好消息：更有经验的 Rust 开发者反映，一旦他们适应所有权系统一段时间之后，与借用检查器的冲突会越来越少。

记住这些之后，让我们来学习有关生命周期的内容。

生命周期

借出一个其它人所有资源的引用可以是很复杂的。例如，想象一下下列操作：

1. 我获取了一个某种资源的句柄
2. 我借给你了一个关于这个资源的引用
3. 我决定不再需要这个资源了，然后释放了它，这时你仍然持有它的引用
4. 你决定使用这个资源

噢！你的引用指向一个无效的资源。这叫做悬垂指针（*dangling pointer*）或者"释放后使用"，如果这个资源是内存的话。这种状况的一个小例子像这样：

```
let r;              // Introduce reference: `r`.
{
    let i = 1;      // Introduce scoped value: `i`.
    r = &i;         // Store reference of `i` in `r`.
}                   // `i` goes out of scope and is dropped.

println!("{}", r);  // `r` still refers to `i`.
```

要修正这个问题的话，我们必须确保第四步永远也不在第三步之后发生。在上面的小例子中 Rust 编译器能够报告问题因为它能识别出函数中不同变量的生命周期。

当我们有一个将引用作为参数的函数时情况就变得更复杂了。考虑如下例子：

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    // ...
#   line
}

let line = "lang:en=Hello World!";
let lang = "en";

let v;
{
    let p = format!("lang:{}=", lang);  // -+ `p` comes into sco
pe.
    v = skip_prefix(line, p.as_str());  //  |
}                                       // -+ `p` goes out of sc
ope.
println!("{}", v);
```

这里我们有个函数 `skip_prefix` ，它获取两个 `&str` 引用作为参数并返回一个 `&str` 引用。通过 `line` 和 `p` 的引用调用它：两个有不同生命周期的变量。现在 `println!` 那行代码的安全依赖于 `skip_prefix` 函数返回的引用是仍然存在的 `line` 还是已经释放掉的 `p` 。

因为存在上述的歧义，Rust 将会拒绝编译示例代码。为了继续我们需要向编译器提供更多关于引用生命周期的信息。这可以通过再函数签名中显式标明生命周期来完成：

```
fn skip_prefix<'a, 'b>(line: &'a str, prefix: &'b str) -> &'a str
 {
    // ...
#   line
}
```

让我们看看所做的修改，但是现在并不深入到语法--之后我们会讲到。第一个修改是再方法名后面加入了 `<'a, 'b>` 。这引入了两个生命周期参数 `'a` 和 `'b` 。接下来函数签名中的每个引用都关联了一个生命周期参数，通过再 `&` 之后加上生命周期的名字。这告诉了编译器不同引用的生命周期是如何关联的。

这样编译器就能推断出 `skip_prefix` 函数的返回值与 `line` 参数有着相同的生命周期，这样就使得之前例子中 `v` 引用即使在 `p` 离开作用域之后也能安全使用。

另外编译器能够检查 `skip_prefix` 返回值的用途，它也能确保之后的实现也遵守函数声明建立的约束。这在实现之后会介绍到的 trait 时显得尤为实用。

注意：认识到生命周期注解是 `_descriptive_` （描述性）而不是 `_prescriptive_` （规定性）是很重要的（译者注：各位可以自行搜索这两个术语）。这意味着引用的生命周期是由代码决定的，而不是由生命周期注解决定的。注解，提供了供编译器用来检查引用的有效性的信息。编译器在简单的情况下无需注解就能进行这种检查，不过在复杂的场景需要程序猿的协助。

语法

`'a` 读作“生命周期 a”。技术上讲，每一个引用都有一些与之相关的生命周期，不过编译器在通常情况让你可以省略（也就是，省略，查看下面的生命周期省略）它们。在我们讲到它之前，让我们看看一个显式生命周期的例子：

```
fn bar<'a>(...)
```

之前我们讨论了一些函数语法，不过我们并没有讨论函数名后面的 `<>` 。一个函数可以在 `<>` 之间有“泛型参数”，生命周期也是其中一种。我们在本书的后面讨论其他类型的泛型。不过现在让我们着重看生命周期。

我们用 `<>` 声明了生命周期。这是说 `bar` 有一个生命周期 `'a` 。如果我们有两个拥有不同生命周期的引用参数，它应该看起来像这样：

```
fn bar<'a, 'b>(...)
```

接着在我们的参数列表中，我们使用了我们命名的生命周期：

```
...(x: &'a i32)
```

如果我们想要一个 `&mut` 引用，我们这么做：

```
...(x: &'a mut i32)
```

如果你对比一下 `&mut i32` 和 `&'a mut i32` ，他们是一样的，只是后者在 `&` 和 `mut i32` 之间夹了一个 `'a` 生命周期。 `&mut i32` 读作“一个 `i32` 的可变引用”，而 `&'a mut i32` 读作“一个带有生命周期'a的i32的可变引用”。

在 `struct` 中

当你处理结构体时你也需要显式的生命周期：

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // This is the same as `let _y = 5; let y = &_y;
`.
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

如你所见， `struct` 也可以有生命周期。跟函数类似的方法，

```
struct Foo<'a> {
# x: &'a i32,
# }
```

声明一个生命周期，接着

```
# struct Foo<'a> {
x: &'a i32,
# }
```

使用它。然而为什么这里我们需要一个生命周期呢？因为我们需要确保任何 `Foo` 的引用不能比它包含的 `i32` 的引用活的更久。

`impl` 块

让我们在 `Foo` 中实现一个方法：

```
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn x(&self) -> &'a i32 { self.x }
}

fn main() {
    let y = &5; // This is the same as `let _y = 5; let y = &_y;
`.
    let f = Foo { x: y };

    println!("x is: {}", f.x());
}
```

如你所见，我们需要在 `impl` 行为 `Foo` 声明一个生命周期。我们重复了 `'a` 两次，就像在函数中： `impl<'a>` 定义了一个生命周期 `'a` ，而 `Foo<'a>` 使用它。

多个生命周期

如果你有多个引用，你可以多次使用同一个生命周期：

```
fn x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str {
#     x
# }
```

这意味着 `x` 和 `y` 存活在同样的作用域内，并且返回值也同样存活在这个作用域内。如果你想要 `x` 和 `y` 有不同的生命周期，你可以使用多个生命周期参数：

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
#     x
# }
```

在这个例子中， `x` 和 `y` 有不同的有效的作用域，不过返回值和 `x` 有相同的生命周期

理解作用域（Thinking in scopes）

理解生命周期的一个办法是想象一个引用有效的作用域。例如：

```
fn main() {
    let y = &5;     // -+ `y` comes into scope.
                    //  |
    // Stuff...     //  |
                    //  |
}                   // -+ `y` goes out of scope.
```

加入我们的 `Foo`：

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ `y` comes into scope.
    let f = Foo { x: y }; // -+ `f` comes into scope.
    // Stuff...           //  |
                          //  |
}                         // -+ `f` and `y` go out of scope.
```

我们的 `f` 生存在 `y` 的作用域之中，所以一切正常。那么如果不是呢？下面的代码不能工作：

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ `x` comes into scope.
                              //  |
    {                         //  |
        let y = &5;           // ---+ `y` comes into scope.
        let f = Foo { x: y }; // ---+ `f` comes into scope.
        x = &f.x;             //  | | This causes an error.
    }                         // ---+ `f` and y go out of scope.
                              //  |
    println!("{}", x);        //  |
}                             // -+ `x` goes out of scope.
```

噢！就像你在这里看到的一样， `f` 和 `y` 的作用域小于 `x` 的作用域。不过当我们尝试 `x = &f.x` 时，我们让 `x` 引用一些将要离开作用域的变量。

命名作用域用来赋予作用域一个名字。有了名字是我们可以谈论它的第一步。

'static

叫做 `static` 的生命周期是特殊的。它代表某样东西具有横跨整个程序的生命周期。大部分 Rust 程序员当他们处理字符串时第一次遇到 `'static` ：

```
let x: &'static str = "Hello, world.";
```

基本字符串是 `&'static str` 类型的因为它的引用一直有效：它们被写入了最终库文件的数据段。另一个例子是全局量：

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

它在二进制文件的数据段中保存了一个 `i32` ，而 `x` 是它的一个引用。

生命周期省略（Lifetime Elision）

Rust支持强大的在函数体中的局部类型推断，不过这在项签名中是禁止的以便允许只通过项签名本身推导出类型。然而，出于人体工程学方面的考虑，有第二个非常限制的叫做“生命周期省略”的推断算法适用于函数签名。它只基于签名部分自身推断而不涉及函数体，只推断生命周期参数，并且只基于3个易于记忆和无歧义的规则，虽然并不隐藏它涉及到的实际类型，因为局部推断可能会适用于它。

当我们讨论生命周期省略的时候，我们使用输入生命周期和输出生命周期（*input lifetime and output lifetime.*）。输入生命周期是关于函数参数的，而输出生命周期是关于函数返回值的。例如，这个函数有一个输入生命周期：

```
fn foo<'a>(bar: &'a str)
```

这个有一个输出生命周期：

```
fn foo<'a>() -> &'a str
```

这个两者皆有：

```
fn foo<'a>(bar: &'a str) -> &'a str
```

有3条规则：

- 每一个被省略的函数参数成为一个不同的生命周期参数。
- 如果刚好有一个输入生命周期，不管是否省略，这个生命周期被赋予所有函数返回值中被省略的生命周期。
- 如果有多个输入生命周期，不过它们当中有一个是 `&self` 或者 `&mut self` ，`self` 的生命周期被赋予所有省略的输出生命周期。

否则，省略一个输出生命周期将是一个错误。

例子

这里有一些省略了生命周期的函数的例子。我们用它们的扩展形式配对了每个省略了生命周期的例子。

```
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded

fn debug(lvl: u32, s: &str); // elided
fn debug<'a>(lvl: u32, s: &'a str); // expanded
```

在上面的例子中，`lvl` 并不需要一个生命周期，因为它不是一个引用（`&`）。只有与引用（例如一个包含引用的`struct`）相关的变量才需要生命周期。

```
fn substr(s: &str, until: u32) -> &str; // elided
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // expanded

fn get_str() -> &str; // ILLEGAL, no inputs

fn frob(s: &str, t: &str) -> &str; // ILLEGAL, two inputs
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // Expanded: Ou
tput lifetime is ambiguous

fn get_mut(&mut self) -> &mut T; // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T; // expanded

fn args<T: ToCStr>(&mut self, args: &[T]) -> &mut Command; // el
ided
fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a m
ut Command; // expanded

fn new(buf: &mut [u8]) -> BufWriter; // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>; // expanded
```

可变性

> mutability.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

可变性，可以改变事物的能力，用在 Rust 中与其它语言有些许不同。可变性的第一方面是它并非默认状态：

```
let x = 5;
x = 6; // Error!
```

我们可以使用 `mut` 关键字来引入可变性：

```
let mut x = 5;

x = 6; // No problem!
```

这是一个可变的变量绑定。当一个绑定是可变的，它意味着你可以改变它指向的内容。所以在上面的例子中，`x` 的值并没有多大的变化，不过这个绑定从一个 `i32` 变成了另外一个。

你也可以使用 `&x` 创建一个它的引用，不过如果你想要使用这个引用来改变它的值，你将会需要一个可变引用：

```
let mut x = 5;
let y = &mut x;
```

`y` 是一个（指向）可变引用的不可变绑定，它意味着你不能把 `y` 与其它变量绑定（`y = &mut z`），不过 `y` 可以用来把 `x` 绑定到别的值上（`*y = 5`）。一个微妙的区别。

当然，如果你想它们都可变：

```
let mut x = 5;
let mut y = &mut x;
```

现在 `y` 可以绑定到另外一个值，并且它引用的值也可以改变。

很重要的一点是 `mut` 是模式的一部分，所以你可以这样做：

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
# }
```

注意这里 `x` 是可变的，`y` 不是。

内部可变性 VS 外部可变性（Interior vs. Exterior Mutability）

然而，当我们谈到 Rust 中什么是“不可变”的时候，它并不意味着它不能被改变：这里是指它的“外部可变性”是不可变的。例如，考虑下 `Arc<T>`：

```
use std::sync::Arc;

let x = Arc::new(5);
let y = x.clone();
```

当我们调用 `clone()` 时，`Arc<T>` 需要更新引用计数。然而你并未使用任何 `mut`，`x` 是一个不可变绑定，并且我们也没有取得 `&mut 5` 或者什么。那么发生了什么呢？

为了解释这些，我们不得不回到Rust指导哲学的核心，内存安全，和Rust用以保证它的机制，所有权系统，和更具体的借用：

> 你可以拥有这两种类型借用的其中一个，但不能同时拥有：
>
> - 拥有 1 个或多个不可变引用（&T）
> - 只有 1 个可变引用（&mut T）

因此，这就是“不可变性”的真正定义：当有两个引用指向同一事物是安全的吗？在 `Arc<T>` 的情况下，是安全的：改变完全包含在结构自身内部。它并不面向用户。为此，它用 `clone()` 分配 `&T`。如果分配 `&mut T` 的话，那么，这将会是一个问题。

其它类型，像std::cell模块中的这一个，则有相反的属性：内部可变性。例如：

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

`RefCell` 使用 `borrow_mut()` 方法来分配它内部资源的 `&mut` 引用。这难道不危险吗？如果我们：

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
# (y, z);
```

事实上这会在运行时引起恐慌。`RefCell` 是这样工作的：它在运行时强制使用Rust的借用规则，并且如果有违反就会 `panic!` 。这让我们绕开了Rust可变性规则的另一方面。让我们先讨论一下它。

字段级别可变性（Field-level mutability）

可变性是借用（ `&mut` ）或者绑定（ `let mut` ）的属性之一。这意味着，例如，你不能让一个结构体的一些字段可变而另一些字段不可变：

```
struct Point {
    x: i32,
    mut y: i32, // Nope.
}
```

结构体的可变性位于它的绑定上：

```
struct Point {
    x: i32,
    y: i32,
}

let mut a = Point { x: 5, y: 6 };

a.x = 10;

let b = Point { x: 5, y: 6};

b.x = 10; // Error: cannot assign to immutable field `b.x`
```

然而，通过使用 `Cell<T>` ，你可以模拟字段级别的可变性：

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
}

let point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);

println!("y: {:?}", point.y);
```

这会打印 `y: Cell { value: 7 }` 。我们成功的更新了 `y` 。

结构体

> structs.md
>
> commit 59e5e65270259666422e51721cc42f261f827386

结构体是一个创建更复杂数据类型的方法。例如，如果我们正在进行涉及到 2D 空间坐标的计算，我们将需要一个 `x` 和一个 `y` 值：

```
let origin_x = 0;
let origin_y = 0;
```

结构体让我们组合它们俩为一个单独，统一的数据类型：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

这里有许多细节，让我们分开说。我们使用了 `struct` 关键字后跟名字来定义了一个结构体。根据传统，结构体使用大写字母开头并且使用驼峰命名法：`PointInSpace` 而不要写成 `Point_In_Space` 。

像往常一样我们用 `let` 创建了一个结构体的实例，不过我们用 `key: value` 语法设置了每个字段。这里顺序不必和声明的时候一致。

最后，因为每个字段都有名字，我们可以访问字段通过圆点记法：`origin.x` 。

结构体中的值默认是不可变的，就像 Rust 中其它的绑定一样。使用 `mut` 使其可变：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    println!("The point is at ({}, {})", point.x, point.y);
}
```

上面的代码会打印 `The point is at (5, 0)` 。

Rust 在语言级别不支持字段可变性，所以你不能像这么写：

```
struct Point {
    mut x: i32, // This causes an error.
    y: i32,
}
```

可变性是绑定的一个属性，不是结构体自身的。如果你习惯于字段级别的可变性，这开始可能看起来有点奇怪，不过这样明显地简化了问题。它甚至可以让你使变量只可变一段临时时间：

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    let point = point; // `point` is now immutable.

    point.y = 6; // This causes an error.
}
```

你的结构体仍然可以包含 `&mut` 指针，它会给你一些类型的可变性：

```
struct Point {
    x: i32,
    y: i32,
}

struct PointRef<'a> {
    x: &'a mut i32,
    y: &'a mut i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    {
        let r = PointRef { x: &mut point.x, y: &mut point.y };

        *r.x = 5;
        *r.y = 6;
    }

    assert_eq!(5, point.x);
    assert_eq!(6, point.y);
}
```

更新语法（Update syntax）

一个包含 `..` 的 `struct` 表明你想要使用一些其它结构体的拷贝的一些值。例如：

```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut point = Point3d { x: 0, y: 0, z: 0 };
point = Point3d { y: 1, .. point };
```

这给了 `point` 一个新的 `y` ，不过保留了 `x` 和 `z` 的值。这也并不必要是同样的 `struct` ，你可以在创建新结构体时使用这个语法，并会拷贝你未指定的值：

```
# struct Point3d {
#     x: i32,
#     y: i32,
#     z: i32,
# }
let origin = Point3d { x: 0, y: 0, z: 0 };
let point = Point3d { z: 1, x: 2, .. origin };
```

元组结构体

Rust 有像另一个元组和结构体的混合体的数据类型。元组结构体有一个名字，不过它的字段没有。他们用 `struct` 关键字声明，并元组前面带有一个名字：

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

这里 `black` 和 `origin` 并不是相同的类型，即使它们有一模一样的值。

元组结构体结构体的成员可以使用点标记或者解构 `let` 访问，就像常规的元组：

```
# struct Color(i32, i32, i32);
# struct Point(i32, i32, i32);
# let black = Color(0, 0, 0);
# let origin = Point(0, 0, 0);
let black_r = black.0;
let Point(_, origin_y, origin_z) = origin;
```

像 `Point(_, origin_y, origin_z)` 这样的模式也可以用于match 表达式。

一个元组结构体非常有用的情况是当他只有一个元素时，我们称之为“新类型（newtype）”模式，因为它允许创建一个区别于它包含的值的类型，同时也标明它的语义：

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

如上所示，通过解构 `let` 可以获取其中的整型值。在这里，`let Inches(integer_length)` 将 `10` 赋值于 `integer_length` 。我们可以用点标记做到同样的事：

```
# struct Inches(i32);
# let length = Inches(10);
let integer_length = length.0;
```

几乎总是可以在使用元组结构体的地方使用 `struct` ，并可能更明确一些。我们可以这样重写 `Color` 和 `Point` ：

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

好的名字是很重要的，同时元组结构体中的值也可以使用点语法被引用， `struct` 提供了真实的名字，而不仅仅是位置。

类单元结构体（Unit-like structs）

你可以定义一个没有任何成员的结构体：

```
struct Electron {} // Use empty braces...
struct Proton;     // ...or just a semicolon.

// Use the same notation when creating an instance.
let x = Electron {};
let y = Proton;
let z = Electron; // Error
```

这样的结构体叫做“类单元”因为它与一个空元组类似，`()`，这有时叫做“单元”。就像一个元组结构体，它定义了一个新类型。

就它本身来看没什么用（虽然有时它可以作为一个标记类型），不过在与其它功能的结合中，它可以变得有用。例如，一个库可能请求你创建一个实现了一个特定特性的结构来处理事件。如果你并不需要在结构中存储任何数据，你可以仅仅创建一个类单元结构体。

枚举

> enums.md
>
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

Rust 中的一个`enum`是一个代表数个可能变量的数据的类型。`enum`中的每个变量都可选是否关联数据：

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

定义变量的语法与用来定义结构体的语法类似：你可以有不带数据的变量（像类单元结构体），带有命名数据的变量，和带有未命名数据的变量（像元组结构体）。然而，不像单独的结构体定义，一个`enum`是一个单独的类型。一个枚举的值可以匹配任何一个变量。因为这个原因，枚举有时被叫做“集合类型”：枚举可能值的集合是每一个变量可能值的集合的总和。

我们使用`::`语法来使用每个变量的名字：它们包含在`enum`名字自身中。这样的话，以下的情况都是可行的：

```
# enum Message {
#     Move { x: i32, y: i32 },
# }
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
    Move { squares: i32 },
    Pass,
}

let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

这两个变量都叫做 `Move` ，不过他们包含在枚举名字中，他们可以无冲突的使用。

枚举类型的一个值包含它是哪个变量的信息，以及任何与变量相关的数据。这有时被作为一个“标记的联合”被提及。因为数据包括一个“标签”表明它的类型是什么。编译器使用这个信息来确保安全的访问枚举中的数据。例如，我们不能简单的尝试解构一个枚举值，就像它是其中一个可能的变体那样：

```
fn process_color_change(msg: Message) {
    let Message::ChangeColor(r, g, b) = msg; // This causes a co
mpile-time error.
}
```

不支持这些操作（比较操作）可能看起来更像限制。不过这是一个我们可以克服的限制。有两种方法：我们自己实现相等（比较），或通过 `match` 表达式模式匹配变量，你会在下一部分学到它。我们还不够了解Rust如何实现相等，不过我们会在特性找到它们。

构造器作为函数（Constructors as functions）

一个枚举的构造器总是可以像函数一样使用。例如：

```
# enum Message {
# Write(String),
# }
let m = Message::Write("Hello, world".to_string());
```

与下面是一样的：

```
# enum Message {
# Write(String),
# }
fn foo(x: String) -> Message {
    Message::Write(x)
}

let x = foo("Hello, world".to_string());
```

这对我们没有什么直接的帮助，直到我们要用到闭包时，这时我们要考虑将函数作为参数传递给其他函数。例如，使用迭代器，我们可以这样把一个 `String` 的vector转换为一个 `Message::Write` 的vector：

```
# enum Message {
# Write(String),
# }

let v = vec!["Hello".to_string(), "World".to_string()];

let v1: Vec<Message> = v.into_iter().map(Message::Write).collect
();
```

匹配

> match.md
>
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

一个简单的 `if` / `else` 往往是不够的，因为你可能有两个或更多个选项。这样 `else` 也会变得异常复杂。Rust 有一个 `match` 关键字，它可以让你有效的取代复杂的 `if` / `else` 组。看看下面的代码：

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

`match` 使用一个表达式然后基于它的值分支。每个分支都是 `val => expression` 这种形式。当匹配到一个分支，它的表达式将被执行。`match` 属于"模式匹配"的范畴，`match` 是它的一个实现。有一个整个关于模式的部分讲到了所有可能的模式。

那么这有什么巨大的优势呢？这确实有优势。第一，`match` 强制穷尽性检查（*exhaustiveness checking*）。你看到了最后那个下划线开头的分支了吗？如果去掉它，Rust 将会给我们一个错误：

```
error: non-exhaustive patterns: `_` not covered
```

Rust 试图告诉我们忘记了一个值。编译器从 `x` 推断它可以是任何 32 位整型值；例如从 -2,147,483,648 到 2,147,483,647。`_` 就像一个匹配所有的分支，它会捕获所有没有被 `match` 分支捕获的所有可能值。如你所见，在上个例子中，我们提供了 1 到 5 的 `mtach` 分支，如果 `x` 是 6 或者其他值，那么它会被 `_` 捕获。

`match` 也是一个表达式，也就是说它可以用在 `let` 绑定的右侧或者其它直接用到表达式的地方：

```
let x = 5;

let number = match x {
    1 => "one",
    2 => "two",
    3 => "three",
    4 => "four",
    5 => "five",
    _ => "something else",
};
```

有时，这是一个把一种类型的数据转换为另一个类型的好方法。

匹配枚举（Matching on enums）

`match` 的另一个重要的作用是处理枚举的可能变量：

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

fn quit() { /* ... */ }
fn change_color(r: i32, g: i32, b: i32) { /* ... */ }
fn move_cursor(x: i32, y: i32) { /* ... */ }

fn process_message(msg: Message) {
    match msg {
        Message::Quit => quit(),
        Message::ChangeColor(r, g, b) => change_color(r, g, b),
        Message::Move { x, y: new_name_for_y } => move_cursor(x,
 new_name_for_y),
        Message::Write(s) => println!("{}", s),
    };
}
```

再一次，Rust编译器检查穷尽性，所以它要求对每一个枚举的变量都有一个匹配分支。如果你忽略了一个，除非你用 `_` 否则它会给你一个编译时错误。

与之前的 `match` 的作用不同，你不能用常规的 `if` 语句来做这些。你可以使用if let语句，它可以被看作是一个 `match` 的简略形式。

模式

> patterns.md
>
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

模式在Rust中十分常见。我们在变量绑定，匹配表达式和其它一些地方使用它们。让我们开始一个快速的关于模式可以干什么的教程！

快速回顾：你可以直接匹配常量，并且 `_` 作为“任何”类型：

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这会打印出 `one` 。

可以在任何分支创建值的绑定：

```
let x = 1;

match x {
    y => println!("x: {} y: {}", x, y),
}
```

这会打印出：

```
x: 1 y: 1
```

注意在同一匹配块中同时拥有捕获全部的 `_` 和捕获全部的绑定会产生错误：

```
let x = 1;

match x {
    y => println!("x: {} y: {}", x, y),
    _ => println!("anything"), // this causes an error as it is
unreachable
}
```

这里有一个模式的陷阱：就像任何引入一个新绑定的语句，他们会引入隐藏。例如：

```
let x = 1;
let c = 'c';

match c {
    x => println!("x: {} c: {}", x, c),
}

println!("x: {}", x)
```

这会打印：

```
x: c c: c
x: 1
```

换句话说，`x =>` 匹配到了模式并引入了一个叫做 `x` 的新绑定。这个新绑定的作用域是匹配分支并拥有 `c` 的值。注意匹配作用域外的 `x` 的值对内部的 `x` 的值并无影响。因为我们已经有了一个 `x`，新的 `x` 隐藏了它。

多重模式（Multiple patterns）

你可以使用 `|` 匹配多个模式：

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

这会输出`one or two`。

解构（Destructuring）

如果你有一个复合数据类型，例如一个结构体，你可以在模式中解构它：

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, y } => println!("({},{})", x, y),
}
```

我们可以用`:`来给出一个不同的名字：

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x1, y: y1 } => println!("({},{})", x1, y1),
}
```

如果你只关心部分值，我们不需要给它们都命名：

```
struct Point {
    x: i32,
    y: i32,
}

let point = Point { x: 2, y: 3 };

match point {
    Point { x, .. } => println!("x is {}", x),
}
```

这会输出 `x is 2` 。

你可以对任何成员进行这样的匹配，不仅仅是第一个：

```
struct Point {
    x: i32,
    y: i32,
}

let point = Point { x: 2, y: 3 };

match point {
    Point { y, .. } => println!("y is {}", y),
}
```

这会输出 `y is 3` 。

这种“解构”行为可以用在任何复合数据类型上，例如元组和枚举

忽略绑定（Ignoring bindings）

你可以在模式中使用 `_` 来忽视它的类型和值。例如，这是一个 `Result<T, E>` 的 `match` ：

```
# let some_value: Result<i32, &'static str> = Err("There was an error");
match some_value {
    Ok(value) => println!("got a value: {}", value),
    Err(_) => println!("an error occurred"),
}
```

在第一个分支，我们绑定了 `Ok` 变量中的值为 `value` ，不过在 `Err` 分支，我们用 `_` 来忽视特定的错误，而只是打印了一个通用的错误信息。

`_` 在任何创建绑定的模式中都有效。这在忽略一个大大结构体的部分字段时很有用：

```
fn coordinate() -> (i32, i32, i32) {
    // Generate and return some sort of triple tuple.
# (1, 2, 3)
}

let (x, _, z) = coordinate();
```

这里，我们绑定元组第一个和最后一个元素为 `x` 和 `z` ，不过省略了中间的元素。

值得注意的是，_ 一开始并不绑定值，这意味着值可能并没有被移动（这里涉及到 Move 和 Copy，应该就是说你不用它的话就不会 Move）：

```
let tuple: (u32, String) = (5, String::from("five"));

// Here, tuple is moved, because the String moved:
let (x, _s) = tuple;

// The next line would give "error: use of partially moved value
: `tuple`".
// println!("Tuple is: {:?}", tuple);

// However,

let tuple = (5, String::from("five"));

// Here, tuple is _not_ moved, as the String was never moved, an
d u32 is Copy:
let (x, _) = tuple;

// That means this works:
println!("Tuple is: {:?}", tuple);
```

这也意味着任何临时变量将会在语句结束时立刻被释放掉：

```
// Here, the String created will be dropped immediately, as it's
 not bound:

let _ = String::from("  hello  ").trim();
```

你也可以在模式中用 `..` 来忽略多个值。

```
enum OptionalTuple {
    Value(i32, i32, i32),
    Missing,
}

let x = OptionalTuple::Value(5, -2, 3);

match x {
    OptionalTuple::Value(..) => println!("Got a tuple!"),
    OptionalTuple::Missing => println!("No such luck."),
}
```

这会打印`Got a tuple!`。

`ref` 和 `ref mut`

如果你想要一个引用，使用`ref`关键字：

```
let x = 5;

match x {
    ref r => println!("Got a reference to {}", r),
}
```

这会输出`Got a reference to 5`。

这里，`match`中的`r`是`&i32`类型的。换句话说，`ref`关键字创建了一个在模式中使用的引用。如果你需要一个可变引用，`ref mut`同样可以做到：

```
let mut x = 5;

match x {
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

范围（Ranges）

你可以用 `...` 匹配一个范围的值：

```
let x = 1;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("anything"),
}
```

这会输出 `one through five` 。

范围经常用在整数和 `char` 上。

```
let x = '';

match x {
    'a' ... 'j' => println!("early letter"),
    'k' ... 'z' => println!("late letter"),
    _ => println!("something else"),
}
```

这会输出 `something else` 。

绑定

你可以使用 `@` 把值绑定到名字上：

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

这会输出 `got a range element 1` 。在你想对一个复杂数据结构进行部分匹配的时候，这个特性十分有用：

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
}

let name = "Steve".to_string();
let x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}"
, a),
    _ => {}
}
```

这会输出 `Some("Steve")` ，因为我们把Person里面的 `name` 绑定到 `a` 。

如果你在使用 `|` 的同时也使用了 `@` ，你需要确保名字在每个模式的每一部分都绑定名字：

```
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element
{}", e),
    _ => println!("anything"),
}
```

守卫（Guards）

你可以用 `if` 来引入匹配守卫（*match guards*）：

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigge
r than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

这会输出`Got an int!`。

如果你在`if`中使用多重模式，`if`条件将适用于所有模式：

```
let x = 4;
let y = false;

match x {
    4 | 5 if y => println!("yes"),
    _ => println!("no"),
}
```

这会打印`no`，因为`if`适用于整个`4 | 5`，而不仅仅是`5`，换句话说，`if`语句的优先级是这样的：

```
(4 | 5) if y => ...
```

而不是这样：

```
4 | (5 if y) => ...
```

混合与匹配（Mix and Match）

(口哨)！根据你的需求，你可以对上面的多种匹配方法进行组合：

```
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

模式十分强大。好好使用它们。

方法语法

> method-syntax.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

函数是伟大的，不过如果你在一些数据上调用了一堆函数，这将是令人尴尬的。 考虑下面代码：

```
baz(bar(foo));
```

我们可以从左向右阅读，我们会看到“baz bar foo”。不过这不是函数被调用的顺序，调用应该是从内向外的：“foo bar baz”。如果能这么做不是更好吗？

```
foo.bar().baz();
```

幸运的是，正如对上面那个问题的猜测，你可以！Rust 通过 `impl` 关键字提供了使用方法调用语法（*method call syntax*）。

方法调用

这是它如何工作的：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());
}
```

这会打印 `12.566371` 。

我们创建了一个代表圆的结构体。我们写了一个 `impl` 块，并且在里面定义了一个方法， `area` 。

方法的第一参数比较特殊， `&self` 。它有3种变体： `self` ， `&self` 和 `&mut self` 。你可以认为这第一个参数就是 `x.foo()` 中的 `x` 。这3种变体对应 `x` 可能的3种类型： `self` 如果它只是栈上的一个值， `&self` 如果它是一个引用，然后 `&mut self` 如果它是一个可变引用。因为我们的 `area` 以 `&self` 作为参数，我们就可以可以像其他参数那样使用它。因为我们知道是一个 `Circle` ，我们可以像任何其他结构体那样访问 `radius` 字段。

我们应该默认使用 `&self` ，就像相比获取所有权你应该更倾向于借用，同样相比获取可变引用更倾向于不可变引用一样。这是一个三种变体的例子：

```rust
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!("taking self by reference!");
    }

    fn mutable_reference(&mut self) {
       println!("taking self by mutable reference!");
    }

    fn takes_ownership(self) {
       println!("taking ownership of self!");
    }
}
```

你可以有任意多个 `impl` 块。上面的例子也可以被写成这样：

```rust
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!("taking self by reference!");
    }
}

impl Circle {
    fn mutable_reference(&mut self) {
       println!("taking self by mutable reference!");
    }
}

impl Circle {
    fn takes_ownership(self) {
       println!("taking ownership of self!");
    }
}
```

链式方法调用（Chaining method calls）

现在我们知道如何调用方法了，例如 `foo.bar()` 。那么我们最开始的那个例子呢， `foo.bar().baz()` ？我们称这个为“方法链”，我们可以通过返回 `self` 来做到这点。

```rust
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }

    fn grow(&self, increment: f64) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius + inc
rement }
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    let d = c.grow(2.0).area();
    println!("{}", d);
}
```

注意返回值：

```rust
# struct Circle;
# impl Circle {
fn grow(&self, increment: f64) -> Circle {
# Circle } }
```

我们看到我们返回了一个 `Circle` 。通过这个函数，我们可以增长一个圆的面积到任意大小。

关联函数（Associated functions）

我们也可以定义一个不带 `self` 参数的关联函数。这是一个 Rust 代码中非常常见的模式：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
}
```

这个关联函数（*associated function*）为我们构建了一个新的 `Circle` 。注意静态函数是通过 `Struct::method()` 语法调用的，而不是 `ref.method()` 语法。

创建者模式（**Builder Pattern**）

我们说我们需要我们的用户可以创建圆，不过我们只允许他们设置他们关心的属性。否则， `x` 和 `y` 将是 `0.0` ，并且 `radius` 将是 `1.0` 。Rust 并没有方法重载，命名参数或者可变参数。我们利用创建者模式来代替。它看起像这样：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}
```

```
impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct CircleBuilder {
    x: f64,
    y: f64,
    radius: f64,
}

impl CircleBuilder {
    fn new() -> CircleBuilder {
        CircleBuilder { x: 0.0, y: 0.0, radius: 1.0, }
    }

    fn x(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.x = coordinate;
        self
    }

    fn y(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.y = coordinate;
        self
    }

    fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
        self.radius = radius;
        self
    }

    fn finalize(&self) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius }
    }
}

fn main() {
    let c = CircleBuilder::new()
                .x(1.0)
```

```
                .y(2.0)
                .radius(2.0)
                .finalize();

    println!("area: {}", c.area());
    println!("x: {}", c.x);
    println!("y: {}", c.y);
}
```

我们在这里又声明了一个结构体， `CircleBuilder` 。我们给它定义了一个创建者函数。我们也在 `Circle` 中定义了 `area()` 方法。我们还定义了另一个方法 `CircleBuilder: finalize()` 。这个方法从构造器中创建了我们最后的 `Circle` 。现在我们使用类型系统来强化我们的考虑：我们可以用 `CircleBuilder` 来强制生成我们需要的 `Circle` 。

字符串

> strings.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

对于每一个程序，字符串都是需要掌握的重要内容。由于Rust主要着眼于系统编程，所以它的字符串处理系统与其它语言有些许区别。每当你碰到一个可变大小的数据结构时，情况都会变得很微妙，而字符串正是可变大小的数据结构。这也就是说，Rust的字符串与一些像C这样的系统编程语言也不相同。

让我们进一步了解一下。一个字符串是一串UTF-8字节编码的Unicode量级值的序列。所有的字符串都确保是有效编码的UTF-8序列。另外，字符串并不以null结尾并且可以包含null字节。

Rust有两种主要的字符串类型： `&str` 和 `String` 。让我们先看看 `&str` 。这叫做字符串片段（*string slices*）。字符串常量是 `&'static str` 类型的：

```
let greeting = "Hello there."; // greeting: &'static str
```

`"Hello there."` 是一个字符串常量而它的类型是 `&'static str` 。字符串常量是静态分配的字符串切片，也就是说它储存在我们编译好的程序中，并且整个程序的运行过程中一直存在。这个 `greeting` 绑定了一个静态分配的字符串的引用。任何接受一个字符串切片的函数也接受一个字符串常量。

字符串常量可以跨多行。有两种形式。第一种会包含新行符和之前的空格：

```
let s = "foo
    bar";

assert_eq!("foo\n    bar", s);
```

第二种，带有 `\` ，会去掉空格和新行符：

```
let s = "foo\
    bar";

assert_eq!("foobar", s);
```

注意通常你不能直接访问一个 `str` ，只能通过 `&str` 引用。这是因为 `str` 是一个不定长类型，它需要额外的运行时信息才能使用。关于更多请查看不定长类型章节。

Rust 当然不仅仅只有 `&str` 。一个 `String` ，是一个在堆上分配的字符串。这个字符串可以增长，并且也保证是UTF-8编码的。 `String` 通常通过一个字符串片段调用 `to_string` 方法转换而来。

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

`String` 可以通过一个 `&` 强制转换为 `&str` ：

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

这种强制转换并不发生在接受 `&str` 的trait而不是 `&str` 本身作为参数的函数上。例如，TcpStream::connect，有一个 `ToSocketAddrs` 类型的参数。 `&str` 可以不用转换不过 `String` 必须使用 `&*` 显式转换。

```
use std::net::TcpStream;

TcpStream::connect("192.168.0.1:3000"); // Parameter is of type
&str.

let addr_string = "192.168.0.1:3000".to_string();
TcpStream::connect(&*addr_string); // Convert `addr_string` to &
str.
```

把 `String` 转换为 `&str` 的代价很小，不过从 `&str` 转换到 `String` 涉及到分配内存。除非必要，没有理由这样做！

索引（Indexing）

因为字符串是有效 UTF-8 编码的，它不支持索引：

```
let s = "hello";

println!("The first letter of s is {}", s[0]); // ERROR!!!
```

通常，用 `[]` 访问一个数组是非常快的。不过，字符串中每个UTF-8编码的字符可以是多个字节，你必须遍历字符串来找到字符串的第N个字符。这个操作的代价相当高，而且我们不想误导读者。更进一步来讲，Unicode实际上并没有定义什么“字符”。我们可以选择把字符串看作一个串独立的字节，或者代码点（codepoints）：

```
let hachiko = "忠犬ハチ公";

for b in hachiko.as_bytes() {
    print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
    print!("{}, ", c);
}

println!("");
```

这会打印出：

```
229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229,
 133, 172,
忠, 犬, ハ, チ, 公,
```

如你所见，这有比 `char` 更多的字节。

你可以这样来获取跟索引相似的东西：

```
# let hachiko = "忠犬ハチ公";
let dog = hachiko.chars().nth(1); // Kinda like `hachiko[1]`.
```

这强调了我们不得不遍历整个 `char` 的列表。

切片（Slicing）

你可以使用切片语法来获取一个字符串的切片：

```
let dog = "hachiko";
let hachi = &dog[0..5];
```

注意这里是字节偏移，而不是字符偏移。所以如下代码在运行时会失败：

```
let dog = "忠犬ハチ公";
let hachi = &dog[0..2];
```

给出如下错误：

```
thread 'main' panicked at 'byte index 2 is not a char boundary;
it is inside '忠'
(bytes 0..3) of `忠犬ハチ公`'
```

连接（Concatenation）

如果你有一个 `String` ，你可以在它后面接上一个 `&str` ：

```
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

不过如果你有两个 `String` ，你需要一个 `&` ：

```
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

这是因为 `&String` 可以自动转换为一个 `&str` 。这个功能叫做 `Deref` 转换。

泛型

> generics.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

有时，当你编写函数或数据类型时，我们可能会希望它能处理多种类型的参数。幸运的是，Rust有一个能给我们更好选择的功能：泛型。泛型在类型理论中叫做参数多态（*parametric polymorphism*），它意味着它们是对于给定参数（parametric）能够有多种形式（ `poly` 是多， `morph` 是形态）的函数或类型。

不管怎么样，类型理论就说这么多，现在我们来看些泛型代码。Rust 标准库提供了一个范型的类型—— `Option<T>` ：

```
enum Option<T> {
    Some(T),
    None,
}
```

之前你已见过几次的 `<T>` 部分代表它是一个泛型数据类型。在上面的枚举声明中，每当我们看到 `T` ，我们用这个类型代替我们泛型中使用的类型。下面是一个使用 `Option<T>` 的例子，它带有额外的类型标注：

```
let x: Option<i32> = Some(5);
```

在类型声明中，我们看到 `Option<i32>` 。注意它与 `Option<T>` 的相似之处。在这个特定的 `Option` 中， `T` 的值为 `i32` 。在绑定的右侧，我们用了 `Some(T)` ，其中 `T` 是 `5` 。因为它是 `i32` 型的，两边类型相符，所以皆大欢喜。如果不相符，我们会得到一个错误：

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected `core::option::Option<f64>`,

// found `core::option::Option<_>` (expected f64 but found integral variable)
```

这并不意味着我们不能写用 `f64` 的 `Option<T>` ！只是类型必须相符：

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

这样就好了。一处定义，到处使用。

不一定只有一个类型是泛型的。想想Rust标准库中另一个类似的 `Result<T, E>` 类型：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

这里有两个泛型类型： `T` 和 `E` 。另外，大写字母可以是任何你喜欢的（大写）字母。我们可以定义 `Result<T, E>` 为：

```
enum Result<A, Z> {
    Ok(A),
    Err(Z),
}
```

如果你想这么做的话。惯例告诉我们第一个泛型参数应该是 `T` ，代表 `type` ，然后我们用 `E` 来代表 `error` 。然而，Rust 并不管这些。

`Result<T, E>` 意图作为计算的返回值，并为了能够在不能工作时返回一个错误。

泛型函数

我们可以用熟悉的语法编写一个获取泛型参数的函数：

```
fn takes_anything<T>(x: T) {
    // Do something with `x`.
}
```

语法有两部分：`<T>`代表“这个函数带有一个泛型类型”，而`x: T`代表“`x`是`T`类型的”。

多个参数可以有相同的泛型类型：

```
fn takes_two_of_the_same_things<T>(x: T, y: T) {
    // ...
}
```

我们可以写一个获取多个（泛型）类型的版本：

```
fn takes_two_things<T, U>(x: T, y: U) {
    // ...
}
```

泛型结构体（Generic structs）

你也可以在一个`struct`中储存泛型类型：

```
struct Point<T> {
    x: T,
    y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

与函数类似，`<T>`是我们声明的泛型参数，而我们也接着在类型定义中使用`x: T`。

当你想要给泛型`struct`增加一个实现时，你可以在`impl`声明类型参数：

```
# struct Point<T> {
#     x: T,
#     y: T,
# }
#
impl<T> Point<T> {
    fn swap(&mut self) {
        std::mem::swap(&mut self.x, &mut self.y);
    }
}
```

目前为止你已经见过了支持几乎任何类型的泛型。他们在很多地方都是有用的：你已经见过了 `Option<T>` ，接下来你还将见到像 `Vec<T>` 这样的通用容器类型。另一方面，通常你想要用灵活性去换取更强的表现力。阅读trait bound章节来了解为什么和如何做。

消除歧义（Resolving ambiguities）

大部分时候当涉及到泛型时，编译器可以自动推断出泛型参数：

```
// v must be a Vec<T> but we don't know what T is yet
let mut v = Vec::new();
// v just got a bool value, so T must be bool!
v.push(true);
// Debug-print v
println!("{:?}", v);
```

但是有的时候，编译器需要一些帮助。例如，如下如果省略最后一行的打印，会得到一个编译错误：

```
let v = Vec::new();
//      ^^^^^^^^ cannot infer type for `T`
//
// note: type annotations or generic parameter binding required
println!("{:?}", v);
```

我们要么可以使用一个类型注解来解决它：

```
let v: Vec<bool> = Vec::new();
println!("{:?}", v);
```

要么通过一个叫做‘turbofish’ `::<>` 的语法来绑定泛型参数 `T` ：

```
let v = Vec::<bool>::new();
println!("{:?}", v);
```

第二种方法在我们并不想要将结果绑定到一个变量时很有用。它也可以用来在函数和方法中绑定泛型参数。查看迭代器与消费者 章节来获取一个例子。

Traits

> traits.md
> commit 47f4de574df32c32b4b14081e5d6e90275b08ed3

trait 是一个告诉 Rust 编译器一个类型必须提供哪些功能语言特性。

你还记得`impl`关键字吗，曾用方法语法调用方法的那个？

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

trait 也很类似，除了我们用函数标记来定义一个 trait，然后为结构体实现 trait。例如，我们为`Circle`实现`HasArea` trait：

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

如你所见，`trait` 块与 `impl` 看起来很像，不过我们没有定义一个函数体，只是函数标记。当我们 `impl` 一个trait时，我们使用 `impl Trait for Item` ，而不是仅仅 `impl Item` 。

`Self` 可以被用在类型标记中表示被作为参数传递的实现了这个 trait 的类型的一个实例。`Self` ，`&Self` 和 `&mut Self` 可以根据所需不同级别的所有权来使用。

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;

    fn is_larger(&self, &Self) -> bool;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }

    fn is_larger(&self, other: &Self) -> bool {
        self.area() > other.area()
    }
}
```

泛型函数的 trait bound（Trait bounds on generic functions）

trait 很有用是因为他们允许一个类型对它的行为提供特定的承诺。泛型函数可以显式的限制（或者叫 bound）它接受的类型。考虑这个函数，它并不能编译：

```
fn print_area<T>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Rust 抱怨道：

```
error: no method named `area` found for type `T` in the current
scope
```

因为 `T` 可以是任何类型，我们不能确定它实现了 `area` 方法。不过我们可以在泛型 `T` 添加一个 trait bound，来确保它实现了对应方法：

```
# trait HasArea {
#     fn area(&self) -> f64;
# }
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

`<T: HasArea>` 语法是指 `any type that implements the HasArea trait`（任何实现了 `HasArea` trait的类型）。因为 trait 定义了函数类型标记，我们可以确定任何实现 `HasArea` 将会拥有一个 `.area()` 方法。

这是一个扩展的例子演示它如何工作：

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
```

```
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 1.0f64,
    };

    print_area(c);
    print_area(s);
}
```

这个程序会输出：

```
This shape has an area of 3.141593
This shape has an area of 1
```

如你所见， `print_area` 现在是泛型的了，并且确保我们传递了正确的类型。如果我们传递了错误的类型：

```
print_area(5);
```

我们会得到一个编译时错误：

```
error: the trait bound `_ : HasArea` is not satisfied [E0277]
```

泛型结构体的 trait bound（Trait bounds on generic structs）

泛型结构体也从 trait bound 中获益。所有你需要做的就是在你声明类型参数时附加上 bound。这里有一个新类型 `Rectangle<T>` 和它的操作 `is_square()` ：

```rust
struct Rectangle<T> {
    x: T,
    y: T,
    width: T,
    height: T,
}

impl<T: PartialEq> Rectangle<T> {
    fn is_square(&self) -> bool {
        self.width == self.height
    }
}

fn main() {
    let mut r = Rectangle {
        x: 0,
        y: 0,
        width: 47,
        height: 47,
    };

    assert!(r.is_square());

    r.height = 42;
    assert!(!r.is_square());
}
```

`is_square()` 需要检查边是相等的，所以边必须是一个实现了 `core::cmp::PartialEq` trait 的类型：

```
impl<T: PartialEq> Rectangle<T> { ... }
```

现在，一个长方形可以用任何可以比较相等的类型定义了。

这里我们定义了一个新的接受任何精度数字的 `Rectangle` 结构体——讲道理，很多类型——只要他们能够比较大小。我们可以对 `HasArea` 结构体， `Square` 和 `Circle` 做同样的事吗？可以，不过他们需要乘法，而要处理它我们需要了解运算符 trait更多。

实现 trait 的规则（Rules for implementing traits）

目前为止，我们只在结构体上添加 trait 实现，不过你可以为任何类型实现一个 trait，比如 `f32` 。

```
trait ApproxEqual {
    fn approx_equal(&self, other: &Self) -> bool;
}
impl ApproxEqual for f32 {
    fn approx_equal(&self, other: &Self) -> bool {
        // Appropriate for `self` and `other` being close to 1.0.

        (self - other).abs() <= ::std::f32::EPSILON
    }
}

println!("{}", 1.0.approx_equal(&1.00000001));
```

这看起来有点像狂野西部（Wild West），不过这还有两个限制来避免情况失去控制。第一是如果 trait 并不定义在你的作用域，它并不能实现。这是个例子：为了进行文件I/O，标准库提供了一个 `Write` trait来为 `File` 增加额外的功能。默认， `File` 并不会有这个方法：

```
let mut f = std::fs::File::create("foo.txt").ok().expect("Couldn’t create foo.txt");
let buf = b"whatever"; // buf: &[u8; 8], a byte string literal.
let result = f.write(buf);
# result.unwrap(); // Ignore the error.
```

这里是错误：

```
error: type `std::fs::File` does not implement any method in scope named `write`
let result = f.write(buf);
               ^~~~~~~~~~
```

我们需要先 `use` 这个 `Write` trait：

```
use std::io::Write;

let mut f = std::fs::File::create("foo.txt").expect("Couldn’t create foo.txt");
let buf = b"whatever";
let result = f.write(buf);
# result.unwrap(); // Ignore the error.
```

这样就能无错误的编译了。

这意味着即使有人做了像给 `i32` 增加函数这样的坏事，它也不会影响你，除非你 `use` 了那个 trait。

这还有一个实现 trait 的限制。要么是 trait 要么是你写实现的类型必须由你定义。更准确的说，它们中的一个必须定义于你编写 `impl` 的相同的 crate 中。关于 Rust 的模块和包系统，详见crate 和模块。

所以，我们可以为 `i32` 实现 `HasArea` trait，因为 `HasArea` 在我们的包装箱中。不过如果我们想为 `i32` 实现 `Float` trait，它是由 Rust 提供的，则无法做到，因为这个 trait 和类型都不在我们的包装箱中。

关于 trait 的最后一点：带有 trait 限制的泛型函数是单态（*monomorphization*）（mono：单一，morph：形式）的，所以它是静态分发（*statically dispatched*）的。这是什么意思？查看trait 对象来了解更多细节。

多 trait bound（Multiple trait bounds）

你已经见过你可以用一个trait限定一个泛型类型参数：

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

如果你需要多于1个限定，可以使用 `+` ：

```
use std::fmt::Debug;

fn foo<T: Clone + Debug>(x: T) {
    x.clone();
    println!("{:?}", x);
}
```

`T` 现在需要实现 `Clone` 和 `Debug` 。

where 从句（Where clause）

编写只有少量泛型和trait的函数并不算太糟，不过当它们的数量增加，这个语法就看起来比较诡异了：

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

函数的名字在最左边，而参数列表在最右边。限制写在中间。

Rust有一个解决方案，它叫“where 从句”：

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn main() {
    foo("Hello", "world");
    bar("Hello", "world");
}
```

`foo()` 使用我们刚才的语法，而 `bar()` 使用 `where` 从句。所有你所需要做的就是在定义参数时省略限制，然后在参数列表后加上一个 `where` 。对于很长的列表，你也可以加上空格：

```
use std::fmt::Debug;

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {

    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

这种灵活性可以使复杂情况变得简洁。

`where` 也比基本语法更强大。例如：

```
trait ConvertTo<Output> {
    fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
    fn convert(&self) -> i64 { *self as i64 }
}

// Can be called with T == i32.
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
    x.convert()
}

// Can be called with T == i64.
fn inverse<T>(x: i32) -> T
        // This is using ConvertTo as if it were "ConvertTo<i64>
".
        where i32: ConvertTo<T> {
    x.convert()
}
```

这突显出了 `where` 从句的额外的功能：它允许限制的左侧可以是任意类型（在这里是 `i32` ），而不仅仅是一个类型参数（比如 `T` ）。在这个例子中， `i32` 必须实现 `ConvertTo<T>` 。不同于定义 `i32` 是什么（因为这是很明显的），这里的 `where` 从句限制了 `T` 。

默认方法（Default methods）

默认方法可以增加在 trait 定义中，如果已经知道通常的实现会定义这个方法。例如， `is_invalid()` 定义为与 `is_valid()` 相反：

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

`Foo` trait 的实现者需要实现 `is_valid()` ，不过并不需要实现 `is_invalid()` 。它会使用默认的行为。你也可以选择覆盖默认行为：

```
# trait Foo {
#     fn is_valid(&self) -> bool;
#
#     fn is_invalid(&self) -> bool { !self.is_valid() }
# }
struct UseDefault;

impl Foo for UseDefault {
    fn is_valid(&self) -> bool {
        println!("Called UseDefault.is_valid.");
        true
    }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn is_valid(&self) -> bool {
        println!("Called OverrideDefault.is_valid.");
        true
    }

    fn is_invalid(&self) -> bool {
        println!("Called OverrideDefault.is_invalid!");
        true // Overrides the expected value of is_invalid()
    }
}

let default = UseDefault;
assert!(!default.is_invalid()); // Prints "Called UseDefault.is_
valid."

let over = OverrideDefault;
assert!(over.is_invalid()); // Prints "Called OverrideDefault.is
_valid."
```

继承（Inheritance）

有时，实现一个trait要求实现另一个trait：

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

`FooBar` 的实现也必须实现 `Foo` ，像这样：

```
# trait Foo {
#     fn foo(&self);
# }
# trait FooBar : Foo {
#     fn foobar(&self);
# }
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

如果我们忘了实现 `Foo` ，Rust 会告诉我们：

```
error: the trait bound `main::Baz : main::Foo` is not satisfied
[E0277]
```

Deriving

重复的实现像 `Debug` 和 `Default` 这样的 trait 会变得很无趣。为此，Rust 提供了一个属性来允许我们让 Rust 为我们自动实现 trait：

```
#[derive(Debug)]
struct Foo;

fn main() {
    println!("{:?}", Foo);
}
```

然而，deriving 限制为一些特定的 trait：

- Clone
- Copy
- Debug
- Default
- Eq
- Hash
- Ord
- PartialEq
- PartialOrd

`Drop`

> drop.md
> commit bd8d27beb54ef2a7bb4162d43006792f9ceae361

现在我们讨论了 trait，让我们看看一个由 Rust 标准库提供的特殊 trait，`Drop`。`Drop` trait 提供了一个当一个值离开作用域后运行一些代码的方法。例如：

```
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Dropping!");
    }
}

fn main() {
    let x = HasDrop;

    // Do stuff.

} // `x` goes out of scope here.
```

当在 `main()` 的末尾 `x` 离开作用域的时候，`Drop` 的代码将会执行。`Drop` 有一个方法，他也叫做 `drop()`。它获取一个 `self` 的可变引用。

就是这样！`Drop` 的机制非常简单，不过这有一些细节。例如，值会以与它们声明相反的顺序被丢弃（dropped）。这是另一个例子：

```
struct Firework {
    strength: i32,
}

impl Drop for Firework {
    fn drop(&mut self) {
        println!("BOOM times {}!!!", self.strength);
    }
}

fn main() {
    let firecracker = Firework { strength: 1 };
    let tnt = Firework { strength: 100 };
}
```

这会输出：

```
BOOM times 100!!!
BOOM times 1!!!
```

`tnt` 在 `firecracker` 之前离开作用域（原文大意：TNT 在爆竹之前爆炸），因为它在之后被声明。后进先出。

那么 `Drop` 有什么好处呢？通常来说，`Drop` 用来清理任何与 `struct` 关联的资源。例如，`Arc<T>` 类型是一个引用计数类型。当 `Drop` 被调用，它会减少引用计数，并且如果引用的总数为0，将会清除底层的值。

if let

> if-let.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

`if let` 允许你合并 `if` 和 `let` 来减少特定类型模式匹配的开销。

例如，让我们假设我们有一些 `Option<T>` 。我们想让它是 `Some<T>` 时在其上调用一个函数，而它是 `None` 时什么也不做。这看起来像：

```
# let option = Some(5);
# fn foo(x: i32) { }
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

我们并不一定要在这使用 `match` ，例如，我们可以使用 `if` ：

```
# let option = Some(5);
# fn foo(x: i32) { }
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

这两种选项都不是特别吸引人。我们可以使用 `if let` 来优雅地完成相同的功能：

```
# let option = Some(5);
# fn foo(x: i32) { }
if let Some(x) = option {
    foo(x);
}
```

如果一个模式匹配成功，它绑定任何值的合适的部分到模式的标识符中，并计算这个表达式。如果模式不匹配，啥也不会发生。

如果你想在模式不匹配时做点其他的，你可以使用 `else` ：

```
# let option = Some(5);
# fn foo(x: i32) { }
# fn bar() { }
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

`while let`

类似的，当你想一直循环，直到一个值匹配到特定的模式的时候，你可以选择使用 `while let` 。使用 `while let` 可以把类似这样的代码：

```
let mut v = vec![1, 3, 5, 7, 11];
loop {
    match v.pop() {
        Some(x) =>  println!("{}", x),
        None => break,
    }
}
```

变成这样的代码：

```
let mut v = vec![1, 3, 5, 7, 11];
while let Some(x) = v.pop() {
    println!("{}", x);
}
```

trait对象

> trait-objects.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

当涉及到多态的代码时，我们需要一个机制来决定哪个具体的版本应该得到执行。这叫做“分发”（dispatch）。大体上有两种形式的分发：静态分发和动态分发。虽然 Rust 喜欢静态分发，不过它也提供了一个叫做“trait 对象”的机制来支持动态分发。

背景

在本章接下来的内容中，我们需要一个 trait 和一些实现。让我们来创建一个简单的 `Foo` 。它有一个返回 `String` 的方法。

```
trait Foo {
    fn method(&self) -> String;
}
```

我们也在 `u8` 和 `String` 上实现了这个 trait：

```
# trait Foo { fn method(&self) -> String; }
impl Foo for u8 {
    fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn method(&self) -> String { format!("string: {}", *self) }
}
```

静态分发

我们可以使用 trait 的限制来进行静态分发：

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}"
, *self) } }
# impl Foo for String { fn method(&self) -> String { format!("st
ring: {}", *self) } }
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

在这里 Rust 用“单态”来进行静态分发。这意味着 Rust 会为 `u8` 和 `String` 分别创建一个特殊版本的的 `do_something()` ，然后将对 `do_something` 的调用替换为这些特殊函数。也就是说，Rust 生成了一些像这样的函数：

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}"
, *self) } }
# impl Foo for String { fn method(&self) -> String { format!("st
ring: {}", *self) } }
fn do_something_u8(x: u8) {
    x.method();
}

fn do_something_string(x: String) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something_u8(x);
    do_something_string(y);
}
```

这样做的一个很大的优点在于：静态分发允许函数被内联调用，因为调用者在编译时就知道它，内联对编译器进行代码优化十分有利。静态分发能提高程序的运行效率，不过相应的也有它的弊端：会导致“代码膨胀”（code bloat）。因为在编译出的二进制程序中，同样的函数，对于每个类型都会有不同的拷贝存在。

此外，编译器也不是完美的并且“优化”后的代码可能更慢。例如，过度的函数内联会导致指令缓存膨胀（缓存控制着我们周围的一切）。这也是为何要谨慎使用 `#[inline]` 和 `#[inline(always)]` 的部分原因。另外一个使用动态分发的原因是，在一些情况下，动态分发更有效率。

然而，常规情况下静态分发更有效率，并且我们总是可以写一个小的静态分发的封装函数来进行动态分发，不过反过来不行，这就是说静态调用更加灵活。因为这个原因标准库尽可能的使用了静态分发。

动态分发

Rust 通过一个叫做“trait 对象”的功能提供动态分发。比如说 `&Foo` 、 `Box<Foo>` 这些就是trait对象。它们是一些值，值中储存实现了特定 trait 的任意类型。它的具体类型只能在运行时才能确定。

从一些实现了特定 `trait` 的类型的指针中，可以从通过转型(casting)（例如， `&x as &Foo` ）或者强制转型(coercing it)（例如，把 `&x` 当做参数传递给一个接收 `&Foo` 类型的函数）来取得trait对象。

这些 trait 对象的强制多态和转型也适用于类似于 `&mut Foo` 的 `&mut T` 以及 `Box<Foo>` 的 `Box<T>` 这样的指针，也就是目前为止我们讨论到的所有指针。强制转型和转型是一样的。

这个操作可以被看作“清除”编译器关于特定类型指针的信息，因此trait对象有时被称为“类型清除”（type erasure）。

回到上面的例子，我们可以使用相同的 trait，通过 trait 对象的转型（casting）来进行动态分发：

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}"
, *self) } }
# impl Foo for String { fn method(&self) -> String { format!("st
ring: {}", *self) } }

fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = 5u8;
    do_something(&x as &Foo);
}
```

或者通过强制转型（by concercing）：

```
# trait Foo { fn method(&self) -> String; }
# impl Foo for u8 { fn method(&self) -> String { format!("u8: {}"
, *self) } }
# impl Foo for String { fn method(&self) -> String { format!("st
ring: {}", *self) } }

fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hello".to_string();
    do_something(&x);
}
```

一个使用trait对象的函数并没有为每个实现了 `Foo` 的类型专门生成函数：它只有一份函数的代码，一般（但不总是）会减少代码膨胀。然而，因为调用虚函数，会带来更大的运行时开销，也会大大地阻止任何内联以及相关优化的进行。

为什么用指针？

和很多托管语言不一样，Rust 默认不用指针来存放数据，因此类型有着不同的大小。在编译时知道值的大小（size），以及了解把值作为参数传递给函数、值在栈上移动、值在堆上分配（或释放）并储存等情况，对于 Rust 程序员来说是很重要的。

对于 `Foo` ，我们需要一个值至少是一个 `String` （24字节）或一个 `u8` （1字节），或者其它crate中可实现了 `Foo` （任意字节）的其他类型。如果值没有使用指针存储，我们无法保证代码能对其他类型正常运作，因为其它类型可以是任意大小的。

用指针来储存值意味着当我们使用 trait 对象时值的大小（size）是无关的，只与指针的大小（size）有关。

表现（Representation）

可以在一个 trait 对象上通过一个特殊的函数指针的记录调用的特性函数通常叫做“虚函数表”（由编译器创建和管理）。

trait 对象既简单又复杂：它的核心表现和设计是十分直观的，不过这有一些难懂的错误信息和诡异行为有待发掘。

让我们从一个简单的，带有 trait 对象的运行时表现开始。 `std::raw` 模块包含与复杂的内建类型有相同结构的结构体，包括trait对象：

```
# mod foo {
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
# }
```

这就是了，一个trait对象就像包含一个“数据”指针和“虚函数表”指针的 `&Foo` 。

数据指针指向 trait 对象保存的数据（某个未知的类型 `T` ），和一个虚表指针指向对应 `T` 的 `Foo` 实现的虚函数表。

一个虚表本质上是一个函数指针的结构体，指向每个函数实现的具体机器码。一个像 `trait_object.method()` 的函数调用会从虚表中取出正确的指针然后进行一个动态调用。例如：

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
    // The compiler guarantees that this function is only called
    // with `x` pointing to a u8.
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
```

```
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 1,
    align: 1,

    // Cast to a function pointer:
    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
    // The compiler guarantees that this function is only called
    // with `x` pointing to a String.
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    // Values for a 64-bit computer, halve them for 32-bit ones
    size: 24,
    align: 8,

    method: call_method_on_String as fn(*const ()) -> String,
};
```

在每个虚表中的 `destructor` 字段指向一个会清理虚表类型的任何资源的函数，对于 `u8` 是普通的，不过对于 `String` 它会释放内存。这对于像 `Box<Foo>` 这类有所有权的trait对象来说是必要的，它需要在离开作用域后清理 `Box` 以及它内部的类型所分配的。 `size` 和 `align` 字段储存需要清除类型的大小和它的对齐需求。

假设我们有一些实现了 `Foo` 的值，那么显式的创建和使用 `Foo` trait对象可能看起来有点像这个（忽略不匹配的类型，它们只是指针而已）：

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // Store the data:
    data: &a,
    // Store the methods:
    vtable: &Foo_for_String_vtable
};

// let y: &Foo = x;
let y = TraitObject {
    // Store the data:
    data: &x,
    // Store the methods:
    vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

对象安全（**Object Safety**）

并不是所有 trait 都可以被用来作为一个 trait 对象。例如，vector 实现了 `Clone`，不过如果我们尝试创建一个 trait 对象：

```
let v = vec![1, 2, 3];
let o = &v as &Clone;
```

我们得到一个错误：

```
error: cannot convert to a trait object because trait `core::clo
ne::Clone` is not object-safe [E0038]
let o = &v as &Clone;
        ^~
note: the trait cannot require that `Self : Sized`
let o = &v as &Clone;
        ^~
```

错误表明 `Clone` 并不是“对象安全的（object-safe）”。只有对象安全的 trait 才能成为 trait 对象。一个对象安全的 trait 需要如下两条为真：

- trait 并不要求 `Self: Sized`
- 所有的方法是对象安全的

那么什么让一个方法是对象安全的呢？每一个方法必须要求 `Self: Sized` 或者如下所有：

- 必须没有任何类型参数
- 必须不使用 `Self`

好的。如你所见，几乎所有的规则都谈到了 `Self` 。一个直观的理解是“除了特殊情况，如果你的 trait 的方法使用了 `Self` ，它就不是对象安全的”。

闭包

> closures.md
> commit 59f76f1a2919641f246da62751eee620b4736acf

有时为了整洁和复用打包一个函数和自由变量（*free variables*）是很有用的。自由变量是指被用在函数中来自函数内部作用域并只用于函数内部的变量。对此，我们用一个新名字“闭包”而且 Rust 提供了大量关于他们的实现，正如我们将看到的。

语法

闭包看起来像这样：

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

我们创建了一个绑定，`plus_one`，并把它赋予一个闭包。闭包的参数位于管道（`|`）之中，而闭包体是一个表达式，在这个例子中，`x + 1`。记住 `{}` 是一个表达式，所以我们也可以拥有包含多行的闭包：

```
let plus_two = |x| {
    let mut result: i32 = x;

    result += 1;
    result += 1;

    result
};

assert_eq!(4, plus_two(2));
```

你会注意到闭包的一些方面与用 `fn` 定义的常规函数有点不同。第一个是我们并不需要标明闭包接收和返回参数的类型。我们可以：

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

不过我们并不需要这么写。为什么呢？基本上，这是出于“人体工程学”的原因。因为为命名函数指定全部类型有助于像文档和类型推断，而闭包的类型则很少有文档因为它们是匿名的，并且并不会产生像推断一个命名函数的类型这样的“远距离错误”。

第二个是语法是相似的，不过有点不同。我会增加空格来使它们看起来更像一点：

```
fn  plus_one_v1   (x: i32) -> i32 { x + 1 }
let plus_one_v2 = |x: i32| -> i32 { x + 1 };
let plus_one_v3 = |x: i32|          x + 1  ;
```

有些小区别，不过仍然是相似的。

闭包及环境

之所以把它称为“闭包”是因为它们“包含在环境中”（close over their environment）。这看起来像：

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

这个闭包，`plus_num`，引用了它作用域中的`let`绑定：`num`。更明确的说，它借用了绑定。如果我们做一些会与这个绑定冲突的事，我们会得到一个错误。比如这个：

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

错误是：

```
error: cannot borrow `num` as mutable because it is also borrowe
d as immutable
    let y = &mut num;
                 ^~~
note: previous borrow of `num` occurs here due to use in closure
; the immutable
  borrow prevents subsequent moves or mutable borrows of `num` u
ntil the borrow
  ends
    let plus_num = |x| x + num;
                   ^~~~~~~~~~~
note: previous borrow ends here
fn main() {
    let mut num = 5;
    let plus_num = |x| x + num;

    let y = &mut num;
}
^
```

一个啰嗦但有用的错误信息！如它所说，我们不能取得一个 `num` 的可变借用因为闭包已经借用了它。如果我们让闭包离开作用域，我们可以：

```
let mut num = 5;
{
    let plus_num = |x: i32| x + num;

} // `plus_num` goes out of scope; borrow of `num` ends.

let y = &mut num;
```

然而，如果你的闭包需要它，Rust会取得所有权并移动环境。这个不能工作：

```
let nums = vec![1, 2, 3];

let takes_nums = || nums;

println!("{:?}", nums);
```

这会给我们：

```
note: `nums` moved into closure environment here because it has
type
  `[closure(()) -> collections::vec::Vec<i32>]`, which is non-co
pyable
let takes_nums = || nums;
```

`Vec<T>` 拥有它内容的所有权，而且由于这个原因，当我们在闭包中引用它时，我们必须取得 `nums` 的所有权。这与我们传递 `nums` 给一个取得它所有权的函数一样。

`move` 闭包

我们可以使用 `move` 关键字强制使我们的闭包取得它环境的所有权：

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

现在，即便关键字是 `move` ，变量遵循正常的移动语义。在这个例子中， `5` 实现了 `Copy` ，所以 `owns_num` 取得一个 `5` 的拷贝的所有权。那么区别是什么呢？

```
let mut num = 5;

{
    let mut add_num = |x: i32| num += x;

    add_num(5);
}

assert_eq!(10, num);
```

那么在这个例子中，我们的闭包取得了一个 `num` 的可变引用，然后接着我们调用了 `add_num` ，它改变了其中的值，正如我们期望的。我们也需要将 `add_num` 声明为 `mut` ，因为我们会改变它的环境。

如果我们改为一个 `move` 闭包，这有些不同：

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;

    add_num(5);
}

assert_eq!(5, num);
```

我们只会得到 `5` 。与其获取一个我们 `num` 的可变借用，我们取得了一个拷贝的所有权。

另一个理解 `move` 闭包的方法：它给出了一个拥有自己栈帧的闭包。没有 `move` ，一个闭包可能会绑定在创建它的栈帧上，而 `move` 闭包则是独立的。例如，这意味着大体上你不能从函数返回一个非 `move` 闭包。

不过在我们讨论获取或返回闭包之前，我们应该更多的了解一下闭包实现的方法。作为一个系统语言，Rust给予你了大量的控制你代码的能力，而闭包也是一样。

闭包实现

Rust 的闭包实现与其它语言有些许不同。它们实际上是trait的语法糖。在此之前你可能要确定已经读过trait章节和trait对象。

都读过了？很好。

理解闭包底层是如何工作的关键有点奇怪：使用 `()` 调用函数，像 `foo()` ，是一个可重载的运算符。到此，其它的一切都会明了。在Rust中，我们使用trait系统来重载运算符。调用函数也不例外。我们有三个 trait 来分别重载：

- `Fn`
- `FnMut`
- `FnOnce`

你会注意到这些 trait 之间的些许区别，不过一个大的区别是 `self` ： `Fn` 获取 `&self` ， `FnMut` 获取 `&mut self` ，而 `FnOnce` 获取 `self` 。这包含了所有3种通过通常函数调用语法的 `self` 。不过我们将它们分在 3 个 trait 里，而不是单独的 1 个。这给了我们大量的对于我们可以使用哪种闭包的控制。

闭包的 `|| {}` 语法是上面 3 个 trait 的语法糖。Rust 将会为了环境创建一个结构体， `impl` 合适的 trait，并使用它。

闭包作为参数（Taking closures as arguments）

现在我们知道了闭包是 trait，我们已经知道了如何接受和返回闭包；就像任何其它的 trait！

这也意味着我们也可以选择静态或动态分发。首先，让我们写一个函数，它接受可调用的参数，调用之，然后返回结果：

```
fn call_with_one<F>(some_closure: F) -> i32
    where F: Fn(i32) -> i32 {

    some_closure(1)
}

let answer = call_with_one(|x| x + 2);

assert_eq!(3, answer);
```

我们传递我们的闭包， `|x| x + 2` ，给 `call_with_one` 。它正做了我们说的：它调用了闭包， `1` 作为参数。

让我们更深层的解析 `call_with_one` 的签名：

```
fn call_with_one<F>(some_closure: F) -> i32
#     where F: Fn(i32) -> i32 {
#     some_closure(1) }
```

我们获取一个参数，而它有类型 `F` 。我们也返回一个 `i32` 。这一部分并不有趣。下一部分是：

```
# fn call_with_one<F>(some_closure: F) -> i32
    where F: Fn(i32) -> i32 {
#   some_closure(1) }
```

因为 `Fn` 是一个trait，我们可以用它限制我们的泛型。在这个例子中，我们的闭包取得一个 `i32` 作为参数并返回 `i32` ，所以我们用泛型限制是 `Fn(i32) -> i32` 。

还有一个关键点在于：因为我们用一个trait限制泛型，它会是单态的，并且因此，我们在闭包中使用静态分发。这是非常简单的。在很多语言中，闭包固定在堆上分配，所以总是进行动态分发。在Rust中，我们可以在栈上分配我们闭包的环境，并静态分发调用。这经常发生在迭代器和它们的适配器上，它们经常取得闭包作为参数。

当然，如果我们想要动态分发，我们也可以做到。trait对象处理这种情况，通常：

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

现在我们取得一个trait对象，一个 `&Fn` 。并且当我们将我们的闭包传递给 `call_with_one` 时我们必须获取一个引用，所以我们使用 `&||` 。

下面是一个使用显式生命周期的闭包的例子。有时你可能需要一个获取这样引用的闭包：

```
fn call_with_ref<F>(some_closure:F) -> i32
    where F: Fn(&i32) -> i32 {

    let value = 0;
    some_closure(&value)
}
```

通常你可以指定闭包的参数的生命周期。我们可以在函数声明上指定它：

```
fn call_with_ref<'a, F>(some_closure:F) -> i32
    where F: Fn(&'a i32) -> i32 {
```

然而这导致了一个问题。当一个函数拥有一个显式生命周期参数，那个生命周期必须跟整个调用这个函数的生命周期一样长。借用检查器会抱怨说 `value` 的生命周期并不够长，因为它只位于声明后在函数体的作用域内。

我们需要的是只为它的参数借用其自己的作用域的闭包，而不是整个外层函数的作用域。为此，我们可以使用更高级的 Trait Bound，使用 `for<...>` 语法：

```
fn call_with_ref<F>(some_closure:F) -> i32
    where F: for<'a> Fn(&'a i32) -> i32 {
```

这会让 rust 编译器找到最小的生命周期来调用闭包并满足借用检查器的规则。我们的函数将能顺利编译：

```
fn call_with_ref<F>(some_closure:F) -> i32
    where F: for<'a> Fn(&'a i32) -> i32 {

    let value = 0;
    some_closure(&value)
}
```

函数指针和闭包

一个函数指针有点像一个没有环境的闭包。因此，你可以传递函数指针给任何期待闭包参数的函数，且能够工作：

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

fn add_one(i: i32) -> i32 {
    i + 1
}

let f = add_one;

let answer = call_with_one(&f);

assert_eq!(2, answer);
```

在这个例子中，我们并不是严格的需要这个中间变量 `f`，函数的名字就可以了：

```
let answer = call_with_one(&add_one);
```

返回闭包（Returning closures）

对于函数式风格代码来说在各种情况返回闭包是非常常见的。如果你尝试返回一个闭包，你可能会得到一个错误。在刚接触的时候，这看起来有点奇怪，不过我们会搞清楚。当你尝试从函数返回一个闭包的时候，你可能会写出类似这样的代码：

```
fn factory() -> (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

编译的时候会给出这一长串相关错误：

```
error: the trait bound `core::ops::Fn(i32) -> i32 : core::marker
::Sized` is not satisfied [E0277]
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> i32` does not have a constant size
known at compile-time
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
error: the trait bound `core::ops::Fn(i32) -> i32 : core::marker
::Sized` is not satisfied [E0277]
let f = factory();
    ^
note: `core::ops::Fn(i32) -> i32` does not have a constant size
known at compile-time
let f = factory();
    ^
```

为了从函数返回一些东西，Rust 需要知道返回类型的大小。不过 `Fn` 是一个 trait，它可以是各种大小(size)的任何东西。比如说，返回值可以是实现了 `Fn` 的任意类型。一个简单的解决方法是：返回一个引用。因为引用的大小(size)是固定的，因此返回值的大小就固定了。因此我们可以这样写：

```
fn factory() -> &(Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

不过这样会出现另外一个错误：

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~~
```

对。因为我们有一个引用，我们需要给它一个生命周期。不过我们的 `factory()` 函数不接收参数，所以省略不能用在这。我们可以使用什么生命周期呢？ `'static` ：

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

不过这样又会出现另一个错误：

```
error: mismatched types:
 expected `&'static core::ops::Fn(i32) -> i32`,
    found `[closure@<anon>:7:9: 7:20]`
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~~
```

这个错误让我们知道我们并没有返回一个 `&'static Fn(i32) -> i32` ，而是返回了一个 `[closure <anon>:7:9: 7:20]` 。等等，什么？

因为每个闭包生成了它自己的环境 `struct` 并实现了 `Fn` 和其它一些东西，这些类型是匿名的。它们只在这个闭包中存在。所以Rust把它们显示为 `closure <anon>` ，而不是一些自动生成的名字。

这个错误也指出了返回值类型期望是一个引用，不过我们尝试返回的不是。更进一步，我们并不能直接给一个对象 `'static` 声明周期。所以我们换一个方法并通过 `Box` 装箱 `Fn` 来返回一个 trait 对象。这个几乎可以成功运行：

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}


let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

这还有最后一个问题：

```
error: closure may outlive the current function, but it borrows
`num`,
which is owned by the current function [E0373]
Box::new(|x| x + num)
         ^~~~~~~~~~~
```

好吧，正如我们上面讨论的，闭包借用他们的环境。而且在这个例子中。我们的环境基于一个栈分配的 `5` ， `num` 变量绑定。所以这个借用有这个栈帧的生命周期。所以如果我们返回了这个闭包，这个函数调用将会结束，栈帧也将消失，那么我们的闭包获得了被释放的内存环境！再有最后一个修改，我们就可以让它运行了：

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

通过把内部闭包变为 `move Fn` ，我们为闭包创建了一个新的栈帧。通过 `Box` 装箱，我们提供了一个已知大小的返回值，并允许它离开我们的栈帧。

通用函数调用语法

> ufcs.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

有时，函数可能有相同的名字。就像下面这些代码：

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
    fn f(&self) { println!("Baz’s impl of Foo"); }
}

impl Bar for Baz {
    fn f(&self) { println!("Baz’s impl of Bar"); }
}

let b = Baz;
```

如果我们尝试调用 `b.f()` ，我们会得到一个错误：

```
error: multiple applicable methods in scope [E0034]
b.f();
  ^~~
note: candidate #1 is defined in an impl of the trait `main::Foo
` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Foo"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait `main::Bar
` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Bar"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

我们需要一个区分我们需要调用哪一函数的方法。这个功能叫做"通用函数调用语法"（universal function call syntax），这看起来像这样：

```
# trait Foo {
#     fn f(&self);
# }
# trait Bar {
#     fn f(&self);
# }
# struct Baz;
# impl Foo for Baz {
#     fn f(&self) { println!("Baz's impl of Foo"); }
# }
# impl Bar for Baz {
#     fn f(&self) { println!("Baz's impl of Bar"); }
# }
# let b = Baz;
Foo::f(&b);
Bar::f(&b);
```

让我们拆开来看。

```
Foo::
Bar::
```

调用的这一半是两个 trait 的类型：`Foo` 和 `Bar` 。这样实际上就区分了这两者：Rust 调用你使用的 trait 里面的方法。

```
f(&b)
```

当我们使用方法语法调用像 `b.f()` 这样的方法时，如果 `f()` 需要 `&self` ，Rust 实际上会自动地把 `b` 借用为 `&self` 。而在这个例子中，Rust 并不会这么做，所以我们需要显式地传递一个 `&b` 。

尖括号形式（Angle-bracket Form）

我们刚才讨论的通用函数调用语法的形式：

```
Trait::method(args);
```

上面的形式其实是一种缩写。这是在一些情况下需要使用的扩展形式：

```
<Type as Trait>::method(args);
```

`<>::` 语法是一个提供类型提示的方法。类型位于 `<>` 中。在这个例子中，类型是 `Type as Trait` ，表示我们想要 `method` 的 `Trait` 版本被调用。在没有二义时 `as Trait` 部分是可选的。尖括号也是一样。因此上面的形式就是一种缩写的形式。

这是一个使用较长形式的例子。

```
trait Foo {
    fn foo() -> i32;
}

struct Bar;

impl Bar {
    fn foo() -> i32 {
        20
    }
}

impl Foo for Bar {
    fn foo() -> i32 {
        10
    }
}

fn main() {
    assert_eq!(10, <Bar as Foo>::foo());
    assert_eq!(20, Bar::foo());
}
```

使用尖括号语法让你可以调用指定 trait 的方法而不是继承到的那个。

包装箱和模块

> crates-and-modules.md
> commit 573222ed32fff4e0cb27be940a97344f339ab98b

当一个项目变大以后，良好的软件工程实践是把它分为一堆较小的部分，再把它们装配到一起。定义良好的接口也非常重要，以使有些功能是私有的，而有些是公有的。Rust 有一个模块系统来帮助我们处理这些工作。

基础术语：包装箱和模块

Rust 有两个不同的术语与模块系统有关：包装箱（*crate*）和模块（*module*）。包装箱是其它语言中库（*library*）或包（*package*）的同义词。因此，“Cargo”则是 Rust 包管理工具的名字：你通过 Cargo 把你的包装箱交付给别人。包装箱可以根据项目的不同，生成可执行文件或库文件。

每个包装箱有一个隐含的根模块（*root module*）包含了该包装箱的代码。你可以在根模块下定义一个子模块树。模块让你可以在包装箱内部为代码分区。

作为一个例子，让我们来创建一个短语（*phrases*）包装箱，它会给我们一些不同语言的短语。为了简单起见，仅有“你好”和“再见”这两种短语，并使用英语和日语作为这些短语的语言。我们采用如下模块布局：

```
                                  +-----------+
                              +---| greetings |
                  +---------+ |   +-----------+
              +---| english |---+
              |   +---------+ |   +-----------+
              |               +---| farewells |
+---------+   |                   +-----------+
| phrases |---+
+---------+   |                   +-----------+
              |               +---| greetings |
              |   +----------+ |   +-----------+
              +---| japanese |--+
                  +----------+ |   +-----------+
                              +---| farewells |
                                  +-----------+
```

在这个例子中，`phrases` 是我们包装箱的名字。剩下所有的都是模块。你可以看到它们组成了一个树，以包装箱为根（即 `phrases` 树的根）分叉出来。

现在我们想要在代码中定义这些模块。首先，用 Cargo 创建一个新包装箱：

```
$ cargo new phrases
$ cd phrases
```

如果你还记得，这会生成一个简单的项目：

```
$ tree .
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs` 是我们包装箱的根，与上面图表中的 `phrases` 对应。

定义模块

我们用 `mod` 关键字来定义我们的每一个模块。让我们把 `src/lib.rs` 写成这样：

```
mod english {
    mod greetings {
    }

    mod farewells {
    }
}

mod japanese {
    mod greetings {
    }

    mod farewells {
    }
}
```

在 `mod` 关键字之后是模块的名字。模块的命名采用Rust其它标识符的命名惯例： `lower_snake_case` 。在大括号中（ `{}` ）是模块的内容。

在 `mod` 中，你可以定义子 `mod` 。我们可以用双冒号（ `::` ）标记访问子模块。我们的4个嵌套模块是 `english::greetings` ， `english::farewells` ， `japanese::greetings` 和 `japanese::farewells` 。因为子模块位于父模块的命名空间中，所以这些不会冲突： `english::greetings` 和 `japanese::greetings` 是不同的，即便它们的名字都是 `greetings` 。

因为这个包装箱的根文件叫做 `lib.rs` ，且没有一个 `main()` 函数。Cargo会把这个包装箱构建为一个库：

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target/debug
build  deps  examples  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrases-<hash>.rlib` 是构建好的包装箱。在我们了解如何使用这个包装箱之前，先让我们把它拆分为多个文件。

多文件包装箱

如果每个包装箱只能有一个文件，这些文件将会变得非常庞大。把包装箱分散到多个文件也非常简单，Rust支持两种方法。

除了这样定义一个模块外：

```
mod english {
    // Contents of our module go here.
}
```

我们还可以这样定义：

```
mod english;
```

如果我们这么做的话，Rust会期望能找到一个包含我们模块内容的 `english.rs` 文件，或者包含我们模块内容的 `english/mod.rs` 文件：

注意在这些文件中，你不需要重新定义这些模块：它们已经由最开始的 `mod` 定义。

使用这两个技巧，我们可以将我们的包装箱拆分为两个目录和七个文件：

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libphrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs` 是我们包装箱的根，它看起来像这样：

```rust
mod english;
mod japanese;
```

这两个定义告诉 Rust 去寻找

- `src/english.rs` 或 `src/english/mod.rs`
- `src/japanese.rs` 或 `src/japanese/mod.rs`

具体根据你的偏好。在我们的例子中，因为我们的模块含有子模块，所以我们选择第二种方式。`src/english/mod.rs` 和 `src/japanese/mod.rs` 都看起来像这样：

```rust
mod greetings;
mod farewells;
```

再一次，这些定义告诉 Rust 去寻找

- `src/english/greetings.rs` 或 `src/english/greetings/mod.rs`
- `src/english/farewells.rs` 或 `src/english/farewells/mod.rs`
- `src/japanese/greetings.rs` 或 `src/japanese/greetings/mod.rs`
- `src/japanese/farewells.rs` 或 `src/japanese/farewells/mod.rs`

因为这些子模块没有自己的子模块，我们选择 `src/english/greetings.rs` 和 `src/japanese/farewells.rs` 。

现在 `src/english/greetings.rs` 和 `src/japanese/farewells.rs` 都是空的。让我们添加一些函数。

在 `src/english/greetings.rs` 添加如下：

```
fn hello() -> String {
    "Hello!".to_string()
}
```

在 `src/english/farewells.rs` 添加如下：

```
fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

在 `src/japanese/greetings.rs` 添加如下：

```
fn hello() -> String {
    "こんにちは".to_string()
}
```

当然，你可以从本文复制粘贴这些内容，或者写点别的东西。事实上你写进去“konnichiwa”对我们学习模块系统并不重要。

在 `src/japanese/farewells.rs` 添加如下：

```
fn goodbye() -> String {
    "さようなら".to_string()
}
```

（这是“Sayōnara”，如果你很好奇的话。）

现在我们在包装箱中添加了一些函数，让我们尝试在别的包装箱中使用它。

导入外部的包装箱

我们有了一个库包装箱。让我们创建一个可执行的包装箱来导入和使用我们的库。

创建一个 `src/main.rs` 文件然后写入如下：（现在它还不能编译）

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings
::hello());
    println!("Goodbye in English: {}", phrases::english::farewel
ls::goodbye());

    println!("Hello in Japanese: {}", phrases::japanese::greetin
gs::hello());
    println!("Goodbye in Japanese: {}", phrases::japanese::farew
ells::goodbye());
}
```

`extern crate` 声明告诉Rust我们需要编译和链接 `phrases` 包装箱。然后我们就可以在这里使用 `phrases` 的模块了。就像我们之前提到的，你可以用双冒号引用子模块和之中的函数。

（注意：当导入像“like-this”名字中包含连字符的 crate时，这样的名字并不是一个有效的 Rust 标识符，它可以通过将连字符变为下划线来转换，所以你应该写成 `extern crate like_this;` ）

另外，Cargo假设 `src/main.rs` 是二进制包装箱的根，而不是库包装箱的。现在我们的包中有两个包装箱： `src/lib.rs` 和 `src/main.rs` 。这种模式在可执行包装箱中非常常见：大部分功能都在库包装箱中，而可执行包装箱使用这个库。这样，其它程序可以只使用我们的库，另外这也是各司其职的良好分离。

现在它还不能很好的工作。我们会得到 4 个错误，它们看起来像：

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:38: 4:72 error: function `hello` is private
src/main.rs:4     println!("Hello in English: {}", phrases::engl
ish::greetings::hello());
                                                   ^~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
phrases/src/main.rs:4:5: 4:76 note: expansion site
```

Rust 默认一切都是私有的。让我们深入了解一下这个。

导出公用接口

Rust 允许你严格的控制你的接口哪部分是公有的，所以它们默认都是私有的。你需要使用 `pub` 关键字，来公开它。让我们先关注 `english` 模块，所以让我们像这样减少 `src/main.rs` 的内容：

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings
::hello());
    println!("Goodbye in English: {}", phrases::english::farewel
ls::goodbye());
}
```

在我们的 `src/lib.rs` ，让我们给 `english` 模块声明添加一个 `pub` ：

```
pub mod english;
mod japanese;
```

然后在我们的 `src/english/mod.rs` 中，加上两个 `pub` ：

```
pub mod greetings;
pub mod farewells;
```

在我们的 `src/english/greetings.rs` 中，让我们在 `fn` 声明中加上 `pub` ：

```
pub fn hello() -> String {
    "Hello!".to_string()
}
```

然后在 `src/english/farewells.rs` 中：

```
pub fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

这样，我们的包装箱就可以编译了，虽然会有警告说我们没有使用 `japanese` 的方法：

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/japanese/greetings.rs:1:1: 3:2 warning: function is never us
ed: `hello`, #[warn(dead_code)] on by default
src/japanese/greetings.rs:1 fn hello() -> String {
src/japanese/greetings.rs:2     "こんにちは".to_string()
src/japanese/greetings.rs:3 }
src/japanese/farewells.rs:1:1: 3:2 warning: function is never us
ed: `goodbye`, #[warn(dead_code)] on by default
src/japanese/farewells.rs:1 fn goodbye() -> String {
src/japanese/farewells.rs:2     "さようなら".to_string()
src/japanese/farewells.rs:3 }
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

现在我们的函数是公有的了，我们可以使用它们。好的！然而， `phrases::english::greetings::hello()` 非常长并且重复。Rust 有另一个关键字用来导入名字到当前空间中，这样我们就可以用更短的名字来引用它们。让我们聊聊 `use` 。

用 `use` 导入模块

Rust有一个 `use` 关键字，它允许我们导入名字到我们本地的作用域中。让我们把 `src/main.rs` 改成这样：

```
extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

这两行 `use` 导入了两个模块到我们本地作用域中，这样我们就可以用一个短得多的名字来引用函数。作为一个传统，当导入函数时，导入模块而不是直接导入函数被认为是一个最佳实践。也就是说，你可以这么做：

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

不过这并不理想。这意味着更加容易导致命名冲突。在我们的小程序中，这没什么大不了的，不过随着我们的程序增长，它将会成为一个问题。如果我们有命名冲突，Rust会给我们一个编译错误。举例来说，如果我们将 `japanese` 的函数设为公有，然后这样尝试：

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust会给我们一个编译时错误：

```
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:5: 4:40 error: a value named `hello` has already b
een imported in this module [E0252]
src/main.rs:4 use phrases::japanese::greetings::hello;
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `phrases`.
```

如果你从同样的模块中导入多个名字，我们不必写多遍。Rust有一个简便的语法：

```
use phrases::english::greetings;
use phrases::english::farewells;
```

我们可以使用这个简写：

```
use phrases::english::{greetings, farewells};
```

使用 `pub use` 重导出

你不仅可以用 `use` 来简化标识符。你也可以在包装箱内用它重导出函数到另一个模块中。这意味着你可以展示一个外部接口可能并不直接映射到内部代码结构。

让我们看个例子。修改 `src/main.rs` 让它看起来像这样：

```
extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());

    println!("Hello in Japanese: {}", japanese::hello());
    println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

然后修改 `src/lib.rs` 公开 `japanese` 模块：

```
pub mod english;
pub mod japanese;
```

接下来，把这两个函数声明为公有，先是 `src/japanese/greetings.rs` ：

```
pub fn hello() -> String {
    "こんにちは".to_string()
}
```

然后是 `src/japanese/farewells.rs` ：

```
pub fn goodbye() -> String {
    "さようなら".to_string()
}
```

最后，修改你的 `src/japanese/mod.rs` 为这样：

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;
mod farewells;
```

`pub use` 声明将这些函数导入到了我们模块结构空间中。因为我们在 `japanese` 模块内使用了 `pub use` ，我们现在有了 `phrases::japanese::hello()` 和 `phrases::japanese::goodbye()` 函数，即使它们的代码在 `phrases::japanese::greetings::hello()` 和 `phrases::japanese::farewells::goodbye()` 函数中。内部结构并不反映外部接口。

这里我们对每个我们想导入到 `japanese` 空间的函数使用了 `pub use` 。我们也可以使用通配符来导入 `greetings` 的一切到当前空间中： `pub use self::greetings::*` 。

那么 `self` 怎么办呢？好吧，默认，`use` 声明是绝对路径，从你的包装箱根目录开始。`self` 则使路径相对于你在结构中的当前位置。有一个更特殊的 `use` 形式：你可以使用 `use super::` 来到达你树中当前位置的上一级。一些同学喜欢把 `self` 看作 `.` 而把 `super` 看作 `..`，它们在许多shell表示为当前目录和父目录。

除了 `use` 之外，路径是相对的：`foo::bar()` 引用一个相对我们位置的 `foo` 中的函数。如果它带有 `::` 前缀，它引用了一个不同的 `foo`，一个从你包装箱根开始的绝对路径。

另外，注意 `pub use` 出现在 `mod` 定义之前。Rust要求 `use` 位于最开始。

构建然后运行：

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese: こんにちは
Goodbye in Japanese: さようなら
```

复杂的导入

Rust 提供了多种高级选项来让你的 `extern crate` 和 `use` 语句变得简洁方便。这是一个例子：

```
extern crate phrases as sayings;

use sayings::japanese::greetings as ja_greetings;
use sayings::japanese::farewells::*;
use sayings::english::{self, greetings as en_greetings, farewell
s as en_farewells};

fn main() {
    println!("Hello in English; {}", en_greetings::hello());
    println!("And in Japanese: {}", ja_greetings::hello());
    println!("Goodbye in English: {}", english::farewells::goodb
ye());
    println!("Again: {}", en_farewells::goodbye());
    println!("And in Japanese: {}", goodbye());
}
```

这里发生了什么？

首先， `extern crate` 和 `use` 都允许重命名导入的项。所以 crate 仍然叫“phrases”，不过这里我们以“sayings”来引用它。类似的，第一个 `use` 语句从 crate 中导入 `japanese::greetings` ，不过作为 `ja_greetings` 而不是简单的 `greetings` 。这可以帮助我们消除来自不同包中相似名字的项的歧义。

第二个 `use` 语句用了一个星号来引入 `sayings::japanese::farewells` 模块中的所有公有符号。如你所见之后我们可以不用模块标识来引用日语的 `goodbye` 函数。这类全局引用要保守使用。需要注意的是它只引入公有符号，哪怕在相同模块的代码中引入。

第三个 `use` 语句需要更多的解释。它使用了“大括号扩展（brace expansion）”来将三条 `use` 语句压缩成了一条（这类语法对曾经写过 Linux shell 脚本的人应该很熟悉）。语句的非压缩形式应该是：

```
use sayings::english;
use sayings::english::greetings as en_greetings;
use sayings::english::farewells as en_farewells;
```

如你所见，大括号压缩了位于同一位置的多个项的 `use` 语句，而且在这里 `self` 指向这个位置。注意：大括号不能与星号嵌套或混合。

const 和 static

> const-and-static.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

Rust 有一个用 `const` 关键字定义常量的方法：

```
const N: i32 = 5;
```

与let绑定不同，你必须标注一个 `const` 的类型。

常量贯穿于整个程序的生命周期。更具体的，Rust 中的常量并没有固定的内存地址。这是因为实际上它们会被内联到用到它们的地方。为此对同一常量的引用并不能保证引用到相同的内存地址。

`static`

Rust 以静态量的方式提供了类似“全局变量”的功能。它们与常量类似，不过静态量在使用时并不内联。这意味着对每一个值只有一个实例，并且位于内存中的固定位置。

这是一个例子：

```
static N: i32 = 5;
```

与let绑定不同，你必须标注一个 `static` 的类型。

静态量贯穿于整个程序的生命周期，因此任何存储在常量中的引用有一个 `'static` 生命周期：

```
static NAME: &'static str = "Steve";
```

可变性

你可以用 `mut` 关键字引入可变性：

```
static mut N: i32 = 5;
```

因为这是可变的，一个线程可能在更新 `N` 同时另一个在读取它，导致内存不安全。因此访问和改变一个 `static mut` 是不安全（unsafe）的，因此必须在 `unsafe` 块中操作：

```
# static mut N: i32 = 5;

unsafe {
    N += 1;

    println!("N: {}", N);
}
```

更进一步，任何存储在 `static` 的类型必须实现 `Sync` 。

初始化

`const` 和 `static` 都要求赋予它们一个值。它们必须只能被赋予一个常量表达式的值。换句话说，你不能用一个函数调用的返回值或任何相似的复合值或在运行时赋值。

我应该用哪个？（Which construct should I use?）

几乎所有时候，如果你可以在两者之间选择，选择 `const` 。实际上你很少需要你的常量关联一个内存位置，而且使用 `const` 允许你不止在在自己的包装箱还可以在下游包装箱中使用像常数扩散这样的优化。

一个常量可以看作一个 C 中的 `#define` ：它有元数据开销但无运行时开销。“我应该在 C 中用一个 #define 还是一个 static 呢？”大体上与在 Rust 你应该用常量还是静态量是一个问题。

属性

> attributes.md
> commit bd8d27beb54ef2a7bb4162d43006792f9ceae361

在 Rust 中声明可以用“属性”标注，它们看起来像：

```
#[test]
# fn foo() {}
```

或像这样：

```
# mod foo {
#![test]
# }
```

这两者的区别是 `!` ，它改变了属性作用的对象：

```
#[foo]
struct Foo;

mod bar {
    #![bar]
}
```

`#[foo]` 作用于下一个项，在这就是 `struct` 声明。 `#![bar]` 作用于包含它的项，在这是 `mod` 声明。否则，它们是一样的。它们都以某种方式改变它们附加到的项的意义。

例如，考虑一个像这样的函数：

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

它被标记为 `#[test]` 。这意味着它是特殊的：当你运行测试，这个函数将会执行。当你正常编译时，它甚至不会被包含进来。这个函数现在是一个测试函数。

属性也可以有附加数据：

```
#[inline(always)]
fn super_fast_fn() {
# }
```

或者甚至是键值：

```
#[cfg(target_os = "macos")]
mod macos_only {
# }
```

Rust属性被用在一系列不同的地方。在参考手册中有一个属性的全表。目前，你不能创建你自己的属性，Rust 编译器定义了它们。

type 别名

> type-aliases.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

`type` 关键字让你定义另一个类型的别名：

```
type Name = String;
```

你可以像一个真正类型那样使用这个类型：

```
type Name = String;

let x: Name = "Hello".to_string();
```

然而要注意的是，这一个别名，完全不是一个新的类型。换句话说，因为 Rust 是强类型的，你可以预期两个不同类型的比较会失败：

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

这给出

```
error: mismatched types:
 expected `i32`,
    found `i64`
(expected i32,
    found i64) [E0308]
     if x == y {
             ^
```

不过，如果我们有一个别名：

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

这会无错误的编译。从任何角度来说，`Num` 类型的值与 `i32` 类型的值都是一样的。

你也可以在泛型中使用类型别名：

```
use std::result;

enum ConcreteError {
    Foo,
    Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

这创建了一个特定版本的 `Result` 类型，它总是有一个 `ConcreteError` 作为 `Result<T, E>` 的 `E` 那部分。这通常用于标准库中创建每个子部分的自定义错误。例如，[`io::Result`]。

类型转换

> casting-between-types.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

Rust，和它对安全的关注，提供了两种不同的在不同类型间转换的方式。第一个， `as` ，用于安全转换。相反， `transmute` 允许任意的转换，而这是 Rust 中最危险的功能之一！

强制转换（Coercion）

类型间的强制转换是隐式的并没有自己的语法，不过可以写作 `as` 。

强转出现在 `let` ， `const` 和 `static` 语句；函数调用参数；结构体初始化的字符值；和函数返回值中。

最常用的强转的例子是从引用中去掉可变性：

- `&mut T` 到 `&T`

一个相似的转换时去掉一个裸指针的可变性：

- `*mut T` 到 `*const T`

引用也能被强转为裸指针：

- `&T` 到 `*const T`
- `&mut T` 到 `*mut T`

自定义强转可以用 `Deref` 定义。

强转是可传递的。

`as`

`as` 关键字进行安全的转换：

```
let x: i32 = 5;

let y = x as i64;
```

有三种形式的安全转换：显式转换，数字类型之间的转换，和指针转换。

转换并不是可传递的：即便是 `e as U1 as U2` 是一个有效的表达式，`e as U2` 也不必要是（事实上只有在 `U1` 强转为 `U2` 时才有效）。

显式转换（Explicit coercions）

`e as U` 是有效的仅当 `e` 是 `T` 类型而且 `T` 能强转为 `U`。

数值转换

`e as U` 的转换在如下情况下也是有效的：

- `e` 是 `T` 类型而且 `T` 和 `U` 是任意数值类型：`numeric-cast`
- `e` 是一个类 C 语言枚举（变量并没有附加值），而且 `U` 是一个整型：`enum-cast`
- `e` 是 `bool` 或 `char` 而且 `T` 是一个整型：`prim-int-cast`
- `e` 是 `u8` 而且 `U` 是 `char`：`u8-char-cast`

例如：

```
let one = true as u8;
let at_sign = 64 as char;
let two_hundred = -56i8 as u8;
```

数值转换的语义是：

- 两个相同大小的整型之间（例如：`i32` -> `u32`）的转换是一个 `no-op`
- 从一个大的整型转换为一个小的整型（例如：`u32` -> `u8`）会截断
- 从一个小的整型转换为一个大的整型（例如：`u8` -> `u32`）会
  - 如果源类型是无符号的会补零（zero-extend）
  - 如果源类型是有符号的会符号（sign-extend）
- 从一个浮点转换为一个整型会向 0 舍入

    - 注意：目前如果舍入的值并不能用目标整型表示的话会导致未定义行为（Undefined Behavior）。这包括 Inf 和 NaN。这是一个 bug 并会被修复。
- 从一个整型转换为一个浮点会产生整型的浮点表示，如有必要会舍入（未指定舍入策略）
- 从 f32 转换为 f64 是完美无缺的
- 从 f64 转换为 f32 会产生最接近的可能值（未指定舍入策略）
    - 注意：目前如果值是有限的不过大于或小于 f32 所能表示的最大最小值会导致未定义行为（Undefined Behavior）。这是一个 bug 并会被修复。

指针转换

你也许会惊讶，裸指针与整型之间的转换是安全的，而且不同类型的指针之间的转换遵循一些限制。只有解引用指针是不安全的：

```
let a = 300 as *const char; // `a` is a pointer to location 300.
let b = a as u32;
```

`e as U` 在如下情况是一个有效的指针转换：

- `e` 是 `*T` 类型，`U` 是 `*U_0` 类型，且要么 `U_0: Sized` 要么 `unsize_kind(T) == unsize_kind(U_0)`：`ptr-ptr-cast`
- `e` 是 `*T` 类型且 `U` 是数值类型，同时 `T: Sized`：`ptr-addr-cast`
- `e` 是一个整型且 `U` 是 `*U_0` 类型，同时 `U_0: Sized`：`addr-ptr-cast`
- `e` 是 `&[T; n]` 类型且 `U` 是 `*const T` 类型：`array-ptr-cast`
- `e` 是函数指针且 `U` 是 `*T` 类型，同时 `T: Sized`：`fptr-ptr-cast`
- `e` 是函数指针且 `U` 是一个整型：`fptr-addr-cast`

`transmute`

`as` 只允许安全的转换，并会拒绝例如尝试将 4 个字节转换为一个 `u32`：

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // Four u8s makes a u32.
```

这个错误为：

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // Four u8s makes a u32.
        ^~~~~~~~
```

这是一个“非标量转换（non-scalar cast）”因为这里我们有多个值：四个元素的数组。这种类型的转换是非常危险的，因为他们假设多种底层结构的实现方式。为此，我们需要一些更危险的东西。

`transmute` 函数由编译器固有功能提供，它做的工作非常简单，不过非常可怕。它告诉Rust对待一个类型的值就像它是另一个类型一样。它这样做并不管类型检查系统，并完全信任你。

在我们之前的例子中，我们知道一个有 4 个 `u8` 的数组可以正常代表一个 `u32`，并且我们想要进行转换。使用 `transmute` 而不是 `as`，Rust 允许我们：

```rust
use std::mem;

fn main() {
    unsafe {
        let a = [0u8, 1u8, 0u8, 0u8];
        let b = mem::transmute::<[u8; 4], u32>(a);
        println!("{}", b); // 256
        // Or, more concisely:
        let c: u32 = mem::transmute(a);
        println!("{}", c); // 256
    }
}
```

为了使它编译通过我们要把这些操作封装到一个 `unsafe` 块中。技术上讲，只有 `mem::transmute` 调用自身需要位于块中，不过在这个情况下包含所有相关的内容是有好处的，这样你就知道该看哪了。在这例子中，`a` 的细节也是重要的，所以它们放到了块中。你会看到各种风格的代码，有时上下文离得太远，因此在 `unsafe` 中包含所有的代码并不是一个好主意。

虽然 `transmute` 做了非常少的检查，至少它确保了这些类型是相同大小的，这个错误：

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u64>(a);
}
```

和：

```
error: transmute called with differently sized types: [u8; 4] (3
2 bits) to u64
(64 bits)
```

除了这些，你可以自行随意转换，只能帮你这么多了！

关联类型

> associated-types.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

关联类型是 Rust 类型系统中非常强大的一部分。它涉及到'类型族'的概念，换句话说，就是把多种类型归于一类。这个描述可能比较抽象，所以让我们深入研究一个例子。如果你想编写一个 `Graph` trait，你需要泛型化两个类型：点类型和边类型。所以你可能会像这样写一个 trait， `Graph<N, E>` ：

```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // Etc.
}
```

虽然这可以工作，不过显得很尴尬，例如，任何需要一个 `Graph` 作为参数的函数都需要泛型化的 `N` ode和 `E` dge类型：

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N)
 -> u32 { ... }
```

我们的距离计算并不需要 `Edge` 类型，所以函数签名中 `E` 只是写着玩的。

我们需要的是对于每一种 `Graph` 类型，都使用一个特定的的 `N` ode和 `E` dge类型。我们可以用关联类型来做到这一点：

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
    // Etc.
}
```

现在，我们使用一个抽象的 `Graph` 了：

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> uint { ... }
```

这里不再需要处理 `E` dge类型了。

让我们更详细的回顾一下。

定义关联类型

让我们构建一个 `Graph` trait。这里是定义：

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

十分简单。关联类型使用 `type` 关键字，并出现在trait体和函数中。

这些 `type` 声明跟函数定义一样。例如，如果我们想 `N` 类型实现 `Display` ，这样我们就可以打印出点类型，我们可以这样写：

```
use std::fmt;

trait Graph {
    type N: fmt::Display;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

实现关联类型

就像任何 trait，使用关联类型的 trait 用 `impl` 关键字来提供实现。下面是一个 `Graph` 的简单实现：

```
# trait Graph {
#     type N;
#     type E;
#     fn has_edge(&self, &Self::N, &Self::N) -> bool;
#     fn edges(&self, &Self::N) -> Vec<Self::E>;
# }
struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
        true
    }

    fn edges(&self, n: &Node) -> Vec<Edge> {
        Vec::new()
    }
}
```

这个可笑的实现总是返回 `true` 和一个空的 `Vec<Edge>` ，不过它提供了如何实现这类 trait 的思路。首先我们需要3个 `struct` ，一个代表图，一个代表点，还有一个代表边。如果使用别的类型更合理，也可以那样做，我们只是准备使用 `struct` 来代表这 3 个类型。

接下来是 `impl` 行，它就像其它任何 trait 的实现。

在这里，我们使用 `=` 来定义我们的关联类型。trait 使用的名字出现在 `=` 的左边，而我们 `impl` 的具体类型出现在右边。最后，我们在函数声明中使用具体类型。

trait 对象和关联类型

这里还有另外一个我们需要讨论的语法：trait对象。如果你试图从一个带有关联类型的 trait 创建一个 trait 对象，像这样：

```
# trait Graph {
#     type N;
#     type E;
#     fn has_edge(&self, &Self::N, &Self::N) -> bool;
#     fn edges(&self, &Self::N) -> Vec<Self::E>;
# }
# struct Node;
# struct Edge;
# struct MyGraph;
# impl Graph for MyGraph {
#     type N = Node;
#     type E = Edge;
#     fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
#         true
#     }
#     fn edges(&self, n: &Node) -> Vec<Edge> {
#         Vec::new()
#     }
# }
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

你会得到两个错误：

```
error: the value of the associated type `E` (from the trait `mai
n::Graph`) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N` (from the trai
t
`main::Graph`) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

我们不能这样创建一个trait对象，因为我们并不知道关联的类型。相反，我们可以这样写：

```
# trait Graph {
#     type N;
#     type E;
#     fn has_edge(&self, &Self::N, &Self::N) -> bool;
#     fn edges(&self, &Self::N) -> Vec<Self::E>;
# }
# struct Node;
# struct Edge;
# struct MyGraph;
# impl Graph for MyGraph {
#     type N = Node;
#     type E = Edge;
#     fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
#         true
#     }
#     fn edges(&self, n: &Node) -> Vec<Edge> {
#         Vec::new()
#     }
# }
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

`N=Node`语法允许我们提供一个具体类型，`Node`，作为`N`类型参数。`E=Edge`也是一样。如果我们不提供这个限制，我们不能确定应该`impl`那个来匹配trait对象。

不定长类型

> unsized-types.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

大部分类型有一个特定的大小，以字节为单位，它们在编译时是已知的。例如，一个 `i32` 是 32 位大，或者 4 个字节。然而，有些类型有益于表达，却没有一个定义的大小。它们叫做“不定长”或者“动态大小”类型。一个例子是 `[T]` 。这个类型代表一个特定数量 `t` 的序列。不过我们并不知道有多少，所以大小是未知的。

Rust 知道几个这样的类型，不过它们有一些限制。这有三个：

1. 我们只能通过指针操作一个不定长类型的实例。 `&[T]` 刚好能正常工作，不过 `[T]` 不行。一个 `&[T]` 能正常工作，不过一个 `[T]` 不行。
2. 变量和参数不能拥有动态大小类型。
3. 只有一个 `struct` 的最后一个字段可能拥有一个动态大小类型；其它字段则不可以拥有动态大小类型。枚举变量不可以用动态大小类型作为数据。

所以为什么这很重要？好吧，因为 `[T]` 只能用在一个指针之后，如果我们没有对不定长类型的语言支持，它将不可能这么写：

```
impl Foo for str {
```

或者

```
impl<T> Foo for [T] {
```

相反，你将不得不这么写：

```
impl Foo for &str {
```

意味深长的是，这个实现将只能用于引用，并且不能用于其它类型的指针。通过 `impl for str` ，所有指针，包括（在一些地方，这里会有bug需要修复）用户自定义的智能指针，可以使用这个 `impl` 。

`?Sized`

如果你想要写一个接受动态大小类型的函数，你可以使用这个特殊的 bound 语法，`?Sized`：

```
struct Foo<T: ?Sized> {
    f: T,
}
```

这个 `?Sized`，读作“`T` 可能是 `Sized` 的”，允许我们匹配固定长度和不定长度的类型。所有泛型类型参数隐式包含 `Sized` bound，所以 `?Sized` 可以被用来移除这个隐式 bound。

运算符与重载

> operators-and-overloading.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

Rust 允许有限形式的运算符重载。特定的运算符可以被重载。要支持一个类型间特定的运算符，你可以实现一个的特定的重载运算符的trait。

例如， `+` 运算符可以通过 `Add` 特性重载：

```
use std::ops::Add;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 0 };
    let p2 = Point { x: 2, y: 3 };

    let p3 = p1 + p2;

    println!("{:?}", p3);
}
```

在 `main` 中，我们可以对我们的两个 `Point` 用 `+` 号，因为我们已经为 `Point` 实现了 `Add<Output=Point>` 。

有一系列可以这样被重载的运算符，并且所有与之相关的trait都在 `std::ops` 模块中。查看它的文档来获取完整的列表。

实现这些特性要遵循一个模式。让我们仔细看看 `Add` ：

```
# mod foo {
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
# }
```

这里总共涉及到 3 个类型：你 `impl Add` 的类型， `RHS` ，它默认是 `Self` ，和 `Output` 。对于一个表达式 `let z = x + y` ， `x` 是 `Self` 类型的， `y` 是 `RHS` ，而 `z` 是 `Self::Output` 类型。

```
# struct Point;
# use std::ops::Add;
impl Add<i32> for Point {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // Add an i32 to a Point and get an f64.
# 1.0
    }
}
```

将允许你这样做：

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

在泛型结构体中使用运算符 trait

现在我们知道了运算符 trait 是如何定义的了，我们可以更通用的定义来自trait 章节的 `HasArea` trait 和 `Square` 结构体：

```
use std::ops::Mul;

trait HasArea<T> {
    fn area(&self) -> T;
}

struct Square<T> {
    x: T,
    y: T,
    side: T,
}

impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy {
    fn area(&self) -> T {
        self.side * self.side
    }
}

fn main() {
    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 12.0f64,
    };

    println!("Area of s: {}", s.area());
}
```

对于 `HasArea` 和 `Square` ，我们声明了一个类型参数 `T` 并取代 `f64` 。 `impl` 则需要更深入的修改：

```
impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy { ... }
```

`area` 方法要求我们可以进行边的乘法，所以我们声明的 `T` 类型必须实现 `std::ops::Mul` 。比如上面提到的 `Add` ， `Mul` 自身获取一个 `Output` 参数：因为我们知道相乘时数字并不会改变类型，我也设定它为 `T` 。 `T` 也必须支持

拷贝，所以 Rust 并不尝试将 `self.side` 移动进返回值。

`Deref` 强制多态

> deref-coercions.md
> commit bd8d27beb54ef2a7bb4162d43006792f9ceae361

标准库提供了一个特殊的特性， `Deref` 。它一般用来重载 `*` ，解引用运算符：

```
use std::ops::Deref;

struct DerefExample<T> {
    value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.value
    }
}

fn main() {
    let x = DerefExample { value: 'a' };
    assert_eq!('a', *x);
}
```

这对编写自定义指针类型很有用。然而，有一个与 `Deref` 相关的语言功能："解引用强制多态（deref coercions）"。规则如下：如果你有一个 `U` 类型，和它的实现 `Deref<Target=T>` ，（那么） `&U` 的值将会自动转换为 `&T` 。这是一个例子：

```
fn foo(s: &str) {
    // Borrow a string for a second.
}

// String implements Deref<Target=str>.
let owned = "Hello".to_string();

// Therefore, this works:
foo(&owned);
```

在一个值的前面用 `&` 号获取它的引用。所以 `owned` 是一个 `String` ，`&owned` 是一个 `&String` ，而因为 `impl Deref<Target=str> for String` ，`&String` 将会转换为 `&str` ，而它是 `foo()` 需要的。

这就是了。这是Rust唯一一个为你进行一个自动转换的地方，不过它增加了很多灵活性。例如，`Rc<T>` 类型实现了 `Deref<Target=T>` ，所以这可以工作：

```
use std::rc::Rc;

fn foo(s: &str) {
    // Borrow a string for a second.
}

// String implements Deref<Target=str>.
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// Therefore, this works:
foo(&counted);
```

我们所做的一切就是把我们的 `String` 封装到了一个 `Rc<T>` 里。不过现在我们可以传递 `Rc<String>` 给任何我们有一个 `String` 的地方。 `foo` 的签名并无变化，不过它对这两个类型都能正常工作。这个例子有两个转换： `&Rc<String>` 转换为 `&String` 接着是 `&String` 转换为 `&str` 。只要类型匹配 Rust 将可以做任意多次这样的转换。

标准库提供的另一个非常通用的实现是：

```
fn foo(s: &[i32]) {
    // Borrow a slice for a second.
}

// Vec<T> implements Deref<Target=[T]>.
let owned = vec![1, 2, 3];

foo(&owned);
```

向量可以 `Deref` 为一个切片。

`Deref` 和方法调用

当调用一个方法时 `Deref` 也会出现。考虑下面的例子：

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = &&Foo;

f.foo();
```

即便 `f` 是 `&&Foo` ，而 `foo` 接受 `&self` ，这也是可以工作的。因为这些都是一样的：

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

一个 `&&&&&&&&&&&&&&&&Foo` 类型的值仍然可以调用 `Foo` 定义的方法，因为编译器会插入足够多的 `*` 来使类型正确。而正因为它插入 `*` ，它用了 `Deref` 。

宏

> macros.md
> commit 7eb4469a002172af9b954d13c35753a55925de7f

到目前为止你已经学到了不少 Rust 提供的抽象和重用代码的工具了。这些代码重用单元有丰富的语义结构。例如，函数有类型签名，类型参数有特性限制并且能重载的函数必须属于一个特定的特性。

这些结构意味着 Rust 核心抽象拥有强大的编译时正确性检查。不过作为代价的是灵活性的减少。如果你识别出一个重复代码的模式，你会发现把它们解释为泛型函数，特性或者任何Rust语义中的其它结构很难或者很麻烦。

宏允许我们在句法水平上进行抽象。宏是一个“展开后的”句法形式的速记。这个展开发生在编译的早期，在任何静态检查之前。因此，宏可以实现很多 Rust 核心抽象不能做到的代码重用模式。

缺点是基于宏的代码更难懂，因为它很少利用 Rust 的内建规则。就像常规函数，一个良好的宏可以在不知道其实现的情况下使用。然而，设计一个良好的宏困难的！另外，在宏中的编译错误更难解释，因为它在展开后的代码上描述问题，不是在开发者使用的代码级别。

这些缺点让宏成了所谓“最后求助于的功能”。这并不是说宏的坏话；只是因为它是 Rust 中需要真正简明，良好抽象的代码的部分。切记权衡取舍。

定义一个宏

你可能见过 `vec!` 宏。用来初始化一个任意数量元素的vector。

```
let x: Vec<u32> = vec![1, 2, 3];
# assert_eq!(x, [1, 2, 3]);
```

这不可能是一个常规函数，因为它可以接受任何数量的参数。不过我们可以想象的到它是这些代码的句法简写：

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
# assert_eq!(x, [1, 2, 3]);
```

我们可以使用宏来实现这么一个简写：实际上

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
# fn main() {
#     assert_eq!(vec![1,2,3], [1, 2, 3]);
# }
```

哇哦，这里有好多新语法！让我们分开来看。

```
macro_rules! vec { ... }
```

这里我们定义了一个叫做 `vec` 的宏，跟用 `fn vec` 定义一个 `vec` 函数很相似。再罗嗦一句，我们通常写宏的名字时带上一个感叹号，例如 `vec!` 。感叹号是调用语法的一部分用来区别宏和常规函数。

匹配

宏通过一系列规则定义，它们是模式匹配的分支。上面我们有：

```
( $( $x:expr ),* ) => { ... };
```

这就像一个 `match` 表达式分支，不过匹配发生在编译时 Rust 的语法树中。最后一个分支（这里只有一个分支）的分号是可选的。`=>` 左侧的“模式”叫匹配器（*matcher*）。它有自己的语法。

`$x:expr` 匹配器将会匹配任何 Rust 表达式，把它的语法树绑定到元变量 `$x` 上。`expr` 标识符是一个片段分类符（*fragment specifier*）。在宏进阶章节（已被本章合并，坐等官方文档更新）中列举了所有可能的分类符。匹配器写在 `$(...)` 中，`*` 会匹配 0 个或多个表达式，表达式之间用逗号分隔。

除了特殊的匹配器语法，任何出现在匹配器中的 Rust 标记必须完全相符。例如：

```
macro_rules! foo {
    (x => $e:expr) => (println!("mode X: {}", $e));
    (y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
    foo!(y => 3);
}
```

将会打印：

```
mode Y: 3
```

而这个：

```
foo!(z => 3);
```

我们会得到编译错误：

```
error: no rules expected the token `z`
```

展开

宏规则的右边是正常的 Rust 语法，大部分是。不过我们可以拼接一些匹配器中的语法。例如最开始的例子：

```
$(
    temp_vec.push($x);
)*
```

每个匹配的 `$x` 表达式都会在宏展开中产生一个单独 `push` 语句。展开中的重复与匹配器中的重复“同步”进行（稍后介绍更多）。

因为 `$x` 已经在表达式匹配中声明了，我们并不在右侧重复 `:expr` 。另外，我们并不将用来分隔的逗号作为重复操作的一部分。相反，我们在重复块中使用一个结束用的分号。

另一个细节： `vec!` 宏的右侧有两对大括号。它们经常像这样结合起来：

```
macro_rules! foo {
    () => {{
        ...
    }}
}
```

外层的大括号是 `macro_rules!` 语法的一部分。事实上，你也可以 `()` 或者 `[]` 。它们只是用来界定整个右侧结构的。

内层大括号是展开语法的一部分。记住， `vec!` 在表达式上下文中使用。要写一个包含多个语句，包括 `let` 绑定，的表达式，我们需要使用块。如果你的宏只展开一个单独的表达式，你不需要内层的大括号。

注意我们从未声明宏产生一个表达式。事实上，直到宏被展开之前我们都无法知道。足够小心的话，你可以编写一个能在多个上下文中展开的宏。例如，一个数据类型的简写可以作为一个表达式或一个模式。

重复（Repetition）

重复运算符遵循两个原则：

1. `$(...)*` 对它包含的所有 `$name` 都执行“一层”重复
2. 每个 `$name` 必须有至少这么多的 `$(...)*` 与其相对。如果多了，它将是多余的。

这个巴洛克宏展示了外层重复中多余的变量。

```
macro_rules! o_O {
    (
        $(
            $x:expr; [ $( $y:expr ),* ]
        );*
    ) => {
        &[ $($( $x + $y ),*),* ]
    }
}

fn main() {
    let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

这就是匹配器的大部分语法。这些例子使用了 `$(...)*` ，它指“0次或多次”匹配。另外你可以用 `$(...)+` 代表“1次或多次”匹配。每种形式都可以包括一个分隔符，分隔符可以使用任何除了 `+` 和 `*` 的符号。

这个系统基于Macro-by-Example（PDF链接）。

卫生（Hygiene）

一些语言使用简单的文本替换来实现宏，它导致了很多问题。例如，这个C程序打印 `13` 而不是期望的 `25` 。

```
#define FIVE_TIMES(x) 5 * x

int main() {
    printf("%d\n", FIVE_TIMES(2 + 3));
    return 0;
}
```

展开之后我们得到 `5 * 2 + 3` ，并且乘法比加法有更高的优先级。如果你经常使用C的宏，你可能知道标准的习惯来避免这个问题，或更多其它的问题。在 Rust 中，你不需要担心这个问题。

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, five_times!(2 + 3));
}
```

元变量 `$x` 被解析成一个单独的表达式节点，并且在替换后依旧在语法树中保持原值。

宏系统中另一个常见的问题是变量捕捉（*variable capture*）。这里有一个C的宏，使用了GNU C 扩展来模拟Rust表达式块。

```
#define LOG(msg) ({ \
    int state = get_log_state(); \
    if (state > 0) { \
        printf("log(%d): %s\n", state, msg); \
    } \
})
```

这是一个非常糟糕的用例：

```
const char *state = "reticulating splines";
LOG(state)
```

它展开为：

```
const char *state = "reticulating splines";
{
    int state = get_log_state();
    if (state > 0) {
        printf("log(%d): %s\n", state, state);
    }
}
```

第二个叫做 `state` 的参数参数被替换为了第一个。当打印语句需要用到这两个参数时会出现问题。

等价的Rust宏则会有理想的表现：

```
# fn get_log_state() -> i32 { 3 }
macro_rules! log {
    ($msg:expr) => {{
        let state: i32 = get_log_state();
        if state > 0 {
            println!("log({}): {}", state, $msg);
        }
    }};
}

fn main() {
    let state: &str = "reticulating splines";
    log!(state);
}
```

这之所以能工作时因为 Rust 有一个卫生宏系统。每个宏展开都在一个不同的语法上下文（*syntax context*）中，并且每个变量在引入的时候都在语法上下文中打了标记。这就好像是 `main` 中的 `state` 和宏中的 `state` 被画成了不同的“颜色”，所以它们不会冲突。

这也限制了宏在被执行时引入新绑定的能力。像这样的代码是不能工作的：

```
macro_rules! foo {
    () => (let x = 3;);
}

fn main() {
    foo!();
    println!("{}", x);
}
```

相反你需要在执行时传递变量的名字，这样它会在语法上下文中被正确标记。

```
macro_rules! foo {
    ($v:ident) => (let $v = 3;);
}

fn main() {
    foo!(x);
    println!("{}", x);
}
```

这对 `let` 绑定和 loop 标记有效，对items无效。所以下面的代码可以编译：

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

递归宏

一个宏展开中可以包含更多的宏，包括被展开的宏自身。这种宏对处理树形结构输入时很有用的，正如这这个（简化了的）HTML简写所展示的那样：

```
# #![allow(unused_must_use)]
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
#   // FIXME(#21826)
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

调试宏代码

运行 `rustc --pretty expanded` 来查看宏展开后的结果。输出表现为一个完整的包装箱，所以你可以把它反馈给 `rustc` ，它会有时会比原版产生更好的错误信息。注意如果在同一作用域中有多个相同名字（不过在不同的语法上下文中）的变量的话 `--pretty expanded` 的输出可能会有不同的意义。这种情况下 `--pretty expanded,hygiene` 将会告诉你有关语法上下文的信息。

`rustc` 提供两种语法扩展来帮助调试宏。目前为止，它们是不稳定的并且需要功能入口（feature gates）。

- `log_syntax!(...)` 会打印它的参数到标准输出，在编译时，并且不“展开”任何东西。
- `trace_macros!(true)` 每当一个宏被展开时会启用一个编译器信息。在展开后使用 `trace_macros!(false)` 来关闭它。

句法要求

即使 Rust 代码中含有未展开的宏，它也可以被解析为一个完整的语法树。这个属性对于编辑器或其它处理代码的工具来说十分有用。这里也有一些关于 Rust 宏系统设计的推论。

一个推论是 Rust 必须确定，当它解析一个宏展开时，宏是否代替了

- 0个或多个项
- 0个或多个方法
- 一个表达式
- 一个语句
- 一个模式

一个块中的宏展开代表一些项，或者一个表达式/语句。Rust 使用一个简单的规则来解决这些二义性。一个代表项的宏展开必须是

- 用大括号界定的，例如 `foo! { ... }`
- 分号结尾的，例如 `foo!(...);`

另一个展开前解析的推论是宏展开必须包含有效的Rust记号。更进一步，括号，中括号，大括号在宏展开中必须是封闭的。例如， `foo!([)` 是不允许的。这让Rust知道宏何时结束。

更正式一点，宏展开体必须是一个记号树（*token trees*）的序列。一个记号树是一系列递归的

- 一个由 `()` ， `[]` 或 `{}` 包围的记号树序列
- 任何其它单个记号

在一个匹配器中，每一个元变量都有一个片段分类符（*fragment specifier*），确定它匹配的哪种句法。

- `ident` ：一个标识符。例如：`x` ，`foo`
- `path` ：一个受限的名字。例如：`T::SpecialA`
- `expr` ：一个表达式。例如：`2 + 2` ；`if true then { 1 } else { 2 }` ；`f(42)`
- `ty` ：一个类型。例如：`i32` ；`Vec<(char, String)>` ；`&T`
- `pat` ：一个模式。例如：`Some(t)` ；`(17, 'a')` ；`_`
- `stmt` ：一个单独语句。例如：`let x = 3`
- `block` ：一个大括号界定的语句序列，或者一个表达式。例如：`{ log(error, "hi"); return 12; }`
- `item` ：一个项。例如：`fn foo() { }` ，`struct Bar`
- `meta` ：一个“元数据项”，可以在属性中找到。例如：`cfg(target_os = "windows")`
- `tt` ：一个单独的记号树

对于一个元变量（metavariable）后面的一个记号有一些额外的规则：

- `expr` 和 `stmt` 变量必须后跟任意一个：`=> , ;`
- `ty` 和 `path` 变量必须后跟任意一个：`=> , = | ; : > [ { as where`
- `pat` 变量必须后跟任意一个：`=> , = | if in`
- 其它变量可以后跟任何记号

这些规则为 Rust 语法提供了一些灵活性以便将来的展开不会破坏现有的宏。

宏系统完全不处理解析模糊。例如，`$($i:ident)* $e:expr` 语法总是会解析失败，因为解析器会被强制在解析 `$i` 和解析 `$e` 之间做出选择。改变展开在它们之前分别加上一个记号可以解决这个问题。在这个例子中，你可以写成 `$(I $i:ident)* E $e:expr` 。

范围和宏导入/导出

宏在编译的早期阶段被展开，在命名解析之前。这有一个缺点是与语言中其它结构相比，范围对宏的作用不一样。

定义和展开都发生在同一个深度优先、字典顺序的包装箱的代码遍历中。那么在模块范围内定义的宏对同模块的接下来的代码是可见的，这包括任何接下来的子 `mod` 项。

一个定义在 `fn` 函数体内的宏，或者任何其它不在模块范围内的地方，只在它的范围内可见。

如果一个模块有 `macro_use` 属性，它的宏在子 `mod` 项之后的父模块也是可见的。如果它的父模块也有 `macro_use` 属性那么在父 `mod` 项之后的祖父模块中也是可见的，以此类推。

`macro_use` 属性也可以出现在 `extern crate` 处。在这个上下文中它控制那些宏从外部包装箱中装载，例如

```
#[macro_use(foo, bar)]
extern crate baz;
```

如果属性只是简单的写成 `#[macro_use]` ，所有的宏都会被装载。如果没有 `#[macro_use]` 属性那么没有宏被装载。只有被定义为 `#[macro_export]` 的宏可能被装载。

装载一个包装箱的宏而不链接到输出，使用 `#[no_link]` 。

一个例子：

```rust
macro_rules! m1 { () => (()) }

// Visible here: `m1`.

mod foo {
    // Visible here: `m1`.

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // Visible here: `m1`, `m2`.
}

// Visible here: `m1`.

macro_rules! m3 { () => (()) }

// Visible here: `m1`, `m3`.

#[macro_use]
mod bar {
    // Visible here: `m1`, `m3`.

    macro_rules! m4 { () => (()) }

    // Visible here: `m1`, `m3`, `m4`.
}

// Visible here: `m1`, `m3`, `m4`.
# fn main() { }
```

当这个库被用 `#[macro_use] extern crate` 装载时，只有 `m2` 会被导入。

Rust参考中有一个宏相关的属性列表。

`$crate` 变量

当一个宏在多个包装箱中使用时会产生另一个困难。来看 `mylib` 定义了

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc_a {
    ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
    ($x:expr) => ( ::mylib::increment($x) )
}
# fn main() { }
```

`inc_a` 只能在 `mylib` 内工作，同时 `inc_b` 只能在库外工作。进一步说，如果用户有另一个名字导入 `mylib` 时 `inc_b` 将不能工作。

Rust（目前）还没有针对包装箱引用的卫生系统，不过它确实提供了一个解决这个问题的变通方法。当从一个叫 `foo` 的包装箱总导入宏时，特殊宏变量 `$crate` 会展开为 `::foo` 。相反，当这个宏在同一包装箱内定义和使用时， `$crate` 将展开为空。这意味着我们可以写

```
#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
# fn main() { }
```

来定义一个可以在库内外都能用的宏。这个函数名字会展开为 `::increment` 或 `::mylib::increment` 。

为了保证这个系统简单和正确， `#[macro_use] extern crate ...` 应只出现在你包装箱的根中，而不是在 `mod` 中。

深入（The deep end）

之前的介绍章节提到了递归宏，但并没有给出完整的介绍。还有一个原因令递归宏是有用的：每一次递归都给你匹配宏参数的机会。

作为一个极端的例子，可以，但极端不推荐，用Rust宏系统来实现一个位循环标记自动机。

```
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));

    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));

    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));

    // Halt on empty data string:
    ( $($ps:tt),* ; )
        => (());
}
```

练习：使用宏来减少上面 `bct!` 宏定义中的重复。

常用宏（Common macros）

这里有一些你会在Rust代码中看到的常用宏。

`panic!`

这个宏导致当前线程恐慌。你可以传给这个宏一个信息通过：

```
panic!("oh no!");
```

`vec!`

`vec!` 的应用遍及本书，所以你可能已经见过它了。它方便创建 `Vec<T>` ：

```
let v = vec![1, 2, 3, 4, 5];
```

它也让你可以用重复值创建vector。例如，100个 `0` ：

```
let v = vec![0; 100];
```

`assert!` 和 `assert_eq!`

这两个宏用在测试中。 `assert!` 获取一个布尔值，而 `assert_eq!` 获取两个值并比较它们。 `true` 就通过， `false` 就 `panic!` 。像这样：

```
// A-ok!

assert!(true);
assert_eq!(5, 3 + 2);

// Nope :(

assert!(5 < 3);
assert_eq!(5, 3);
```

`try!`

`try!` 用来进行错误处理。它获取一些可以返回 `Result<T, E>` 的数据，并返回 `T` 如果它是 `Ok<T>` ，或 `return` 一个 `Err(E)` 如果出错了。像这样：

```rust
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = try!(File::create("foo.txt"));

    Ok(())
}
```

它比这么写要更简明：

```rust
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = File::create("foo.txt");

    let f = match f {
        Ok(t) => t,
        Err(e) => return Err(e),
    };

    Ok(())
}
```

`unreachable!`

这个宏用于当你认为一些代码不应该被执行的时候：

```rust
if false {
    unreachable!();
}
```

有时，编译器可能会让你编写一个你认为将永远不会执行的不同分支。在这个例子中，用这个宏，这样如果最终你错了，你会为此得到一个 `panic!` 。

```
let x: Option<i32> = None;

match x {
    Some(_) => unreachable!(),
    None => println!("I know x is None!"),
}
```

`unimplemented!`

`unimplemented!` 宏可以被用来当你尝试去让你的函数通过类型检查，同时你又不想操心去写函数体的时候。一个这种情况的例子是实现一个要求多个方法的特性，而你只想一次搞定一个。用 `unimplemented!` 定义其它的直到你准备好去写它们了。

宏程序（Procedural macros）

如果Rust宏系统不能做你想要的，你可能想要写一个编译器插件。与 `macro_rules!` 宏相比，它能做更多的事，接口也更不稳定，并且bug将更难以追踪。相反你得到了可以在编译器中运行任意Rust代码的灵活性。为此语法扩展插件有时被称为宏程序（*procedural macros*）。

> 实际上. `vec!` 在 libcollections 中的实际定义跟这里的表现并不相同，出于效率和复用的考虑。↩

裸指针

> raw-pointers.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

Rust 的标准库中有一系列不同的智能指针类型，不过这有两个类型是十分特殊的。Rust 的安全大多来源于编译时检查，不过裸指针并没有这样的保证，使用它们是 `unsafe` 的。

`*const T` 和 `*mut T` 在 Rust 中被称为“裸指针”。有时当编写特定类型的库时，为了某些原因你需要绕过 Rust 的安全保障。在这种情况下，你可以使用裸指针来实现你的库，同时暴露一个安全的接口给你的用户。例如， `*` 指针允许别名，允许用来写共享所有权类型，甚至是内存安全的共享内存类型（ `Rc<T>` 和 `Arc<T>` 类型都是完全用 Rust 实现的）。

有一些你需要记住的裸指针不同于其它指针的地方。它们是：

- 不能保证指向有效的内存，甚至不能保证是非空的（不像 `Box` 和 `&` ）；
- 没有任何自动清除，不像 `Box` ，所以需要手动管理资源；
- 是普通旧式类型，也就是说，它不移动所有权，这又不像 `Box` ，因此Rust编译器不能保证不出像释放后使用这种bug；
- ~~被认为是可发送的（如果它的内容是可发送的），因此编译器不能提供帮助确保它的使用是线程安全的；例如，你可以从两个线程中并发的访问 `*mut i32` 而不用同步。~~
- 缺少任何形式的生命周期，不像 `&` ，因此编译器不能判断出悬垂指针；
- 除了不允许直接通过 `*const T` 改变外，没有别名或可变性的保障。

基础

创建一个裸指针是非常安全的：

```rust
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

然而，解引用它则不行。这个并不能工作：

```
let x = 5;
let raw = &x as *const i32;

println!("raw points at {}", *raw);
```

它给出这个错误：

```
error: dereference of raw pointer requires unsafe function or bl
ock [E0133]
     println!("raw points at {}", *raw);
                                  ^~~~
```

当你解引用一个裸指针，你要为它并不指向正确的地方负责。为此，你需要 `unsafe` ：

```
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

println!("raw points at {}", points_at);
```

关于裸指针的更多操作，查看它们的API文档。

FFI

裸指针在FFI中很有用：Rust的 `*const T` 和 `*mut T` 分别与C中的 `const T*` 和 `T*` 类似。关于它们的应用，查看FFI章节。

引用和裸指针

在运行时，指向一份相同数据的裸指针 `*` 和引用有相同的表现。事实上，在安全代码中 `&T` 引用会隐式的转换为一个 `*const T` 同时它们的 `mut` 变体也有类似的行为（这两种转换都可以显式执行，分别为 `value as *const T` 和 `value as *mut T` ）。

反其道而行之，从 `*const` 到 `&` 引用，是不安全的。一个 `&T` 总是有效的，所以，最少， `*const T` 裸指针必须指向一个 `T` 的有效实例。进一步，结果指针必须满足引用的别名和可变性法则。编译器假设这些属性对任何引用都是有效的，不管它们是如何创建的，因而所以任何从裸指针来的转换都断言它们成立。程序员必须保证它。

推荐的转换方法是

```
// Explicit cast:
let i: u32 = 1;
let p_imm: *const u32 = &i as *const u32;

// Implicit coercion:
let mut m: u32 = 2;
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

与使用 `transmute` 相比更倾向于 `&*x` 解引用风格。 `transmute` 远比需要的强大，并且（解引用）更受限的操作会更难以错误使用；例如，它要求 `x` 是一个指针（不像 `transmute` ）。

不安全代码

> unsafe.md
> commit 98b18fabddaa942c6bbbb44509b7c00f4b07915f

Rust 主要魅力是它强大的静态行为保障。不过安全检查天性保守：有些程序实际上是安全的，不过编译器不能验证它是否是真的。为了写这种类型的程序，我们需要告诉编译器稍微放松它的限制。为此，Rust 有一个关键字， `unsafe` 。使用 `unsafe` 的代码比正常代码有更少的限制。

让我们过一遍语法，接着我们讨论语义。 `unsafe` 用在两个上下文中。第一个标记一个函数为不安全的：

```
unsafe fn danger_will_robinson() {
    // Scary stuff...
}
```

例如所有从FFI调用的函数都必须标记为 `unsafe` 。第二个 `unsafe` 的用途是一个不安全块。

```
unsafe {
    // Scary stuff...
}
```

第三个是不安全 trait：

```
unsafe trait Scary { }
```

而第四个是 `impl` 这些 trait：

```
# unsafe trait Scary { }
unsafe impl Scary for i32 {}
```

显式勾勒出那些可能会有 bug 并造成大问题的代码是很重要的。如果一个 Rust 程序段错误了，你可以确认它位于标记为 `unsafe` 部分的什么地方。

“安全”指什么？（What does ‘safe’ mean?）

安全，在 Rust 的上下文中，意味着“不做任何不安全的事”。不过也要明白，有一些特定的行为在你的代码中可能并不合意，但很明显并不是不安全的：

- 死锁
- 内存或其他资源的泄露
- 退出但未调用析构函数
- 整型溢出

Rust 不能避免所有类型的软件错误。有 bug 的代码可能并将会出现在 Rust 中。这些事并不很光彩，不过它们并不特别的定义为 `unsafe` 。

另外，如下列表全是 Rust 中的未定义行为，并且必须被避免，即便在编写 `unsafe` 代码时：

- 数据竞争
- 解引用一个空/悬垂裸指针
- 读 `undef` （未初始化）内存
- 使用裸指针打破指针重叠规则（pointer aliasing rules）
- `&mut T` 和 `&T` 遵循LLVM范围的 `noalias` 模型，除了如果 `&T` 包含一个 `UnsafeCell<U>` 的话。不安全代码必须不能违反这些重叠（aliasing）保证
- 不使用 `UnsafeCell<U>` 改变一个不可变值/引用
- 通过编译器固有功能调用未定义行为：
  - 使用 `std::ptr::offset` （ `offset` 功能）来索引超过对象边界的值，除了允许的末位超出一个字节
  - 在重叠（overlapping）缓冲区上使用 `std::ptr::copy_nonoverlapping_memory` （ `memcpy32/memcpy64` 功能）
- 原生类型的无效值，即使是在私有字段/本地变量中：
  - 空/悬垂引用或装箱
  - `bool` 中一个不是 `false` （ `0` ）或 `true` （ `1` ）的值
  - `enum` 中一个并不包含在类型定义中判别式
  - `char` 中一个代理字（surrogate）或超过 `char::MAX` 的值
  - `str` 中非UTF-8字节序列
- 在外部代码中使用Rust或在Rust中使用外部语言

不安全的超级力量（Unsafe Superpowers）

在不安全函数和不安全块，Rust 将会让你做 3 件通常你不能做的事：只有 3 件。它们是：

1. 访问和更新一个静态可变变量
2. 解引用一个裸指针
3. 调用不安全函数。这是最 NB 的能力

这就是全部。注意到 `unsafe` 不能（例如）“关闭借用检查”是很重要的。为随机的Rust代码加上 `unsafe` 并不会改变它的语义，它并不会开始接受任何东西。

不过确实它会让你写的东西打破一些规则。让我们按顺序过一遍这3个能力。

访问和更新一个 `static mut`

Rust 有一个叫 `static mut` 的功能，它允许改变全局状态。这么做可能造成一个数据竞争，所以它天生是不安全的。关于更多细节，查看静态量部分。

解引用一个裸指针

裸指针让你做任意的指针算数，并会产生一系列不同的内存安全（safety & security）问题。在某种意义上，解引用一个任意指针的能力是你可以做的最危险的事之一。更多关于裸指针，查看它的部分。

调用不安全函数

最后的能力能用于 `unsafe` 的两个方面：你只能在一个不安全块中调用被标记为 `unsafe` 的函数。

这个能力是强力和多变的。Rust 暴露了一些作为不安全函数的编译器固有功能，并且一些不安全函数绕开了安全检查，用安全换速度。

我在重复一遍：即便你可以在一个不安全块和函数中做任何事并不意味你应该这么做。编译器会表现得像你在保持它不变一样（The compiler will act as though you’re upholding its invariants），所以请小心。

高效 Rust

> effective-rust.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

那么你已经学会了如何写一些 Rust 代码了。不过能写一些 Rust 代码和能写好 Rust 代码还是有区别的。

这个部分包含一些相对独立的教程，它们向你展示如何将你的 Rust 带入下一个等级。常见模式和标准库功能将被介绍。你可以选择任意顺序阅读这一部分。

栈和堆

> the-stack-and-the-heap.md
>
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

作为一个系统语言，Rust 在底层运作。如果你有一个高级语言的背景，这可能有一些你不太熟悉的系统编程方面的内容。最重要的一个是内存如何工作，通过栈和堆。如果你熟悉类 C 语言是如何使用栈分配的，这个章节将是一个复习。如果你不太了解，你将会学到这个更通用的概念，不过是专注于 Rust 的。

对于大部分内容，当学习他们的时候，我们会使用一个简化的模型来开始。这使得你可以掌握基础，而不会现在就因为不相关的细节而停滞不前。这里我们使用的例子不一定百分之百准确，不过它代表了目前我们的尝试学习的计数等级。一旦你掌握了基础，学习了更多分配器是如何实现的，虚拟内存和其他高级主题之后，将会揭示出这个特定抽象的漏洞。

内存管理

这两个术语是关于内存管理的。栈和堆是帮助你决定何时分配和释放内存的抽象（概念）。

这是一个高层次的比较：

栈非常快速，并且是 Rust 默认分配内存的地方。不过这个分配位于函数调用的本地，并有大小限制。堆，相反，更慢，并且需要被你的程序显式分配。不过它无事实上的大小限制，并且是全局可访问的。注意这里堆的意义，它意味着已任意顺序分配任意大小的内存块，这与堆数据结构有很大不同（黑我大 Java？）。

栈

让我们讨论下这个 Rust 程序：

```
fn main() {
    let x = 42;
}
```

这个程序有一个变量绑定， `x` 。这个内存需要在什么地方被分配？Rust 默认“栈分配”，也就意味着基本（类型）值“出现在栈上”。这意味着什么呢？

好吧，当函数被调用时，一些内存被分配给所有它的本地变量和一些其它信息。这叫做一个“栈帧（stack frame）”，而为了这个教程的目的，我们将忽略这些额外信息并仅仅考虑我们分配的局部变量。所以在这个例子中，当 `main()` 运行时，我们将为我们的栈帧分配一个单独的32位整型。如你所见，这会自动为你处理，我们并不必须写任何特殊的 Rust 代码或什么的。

当这个函数结束时，它的栈帧被释放。这也是自动发生的，在这里我们也不必要做任何特殊的事情。

这就是关于这个简单程序的一切。在这里你需要理解的关键是栈的分配非常快。因为我们知道所有的局部变量是预先分配的，我们可以一次获取所有的内存。并且因为我们也会同时把它们都扔了，我们可以快速的释放它们。

缺点是如果我们需要它们活过一个单独的函数调用，我们并不能保留它们的值。我们也还没有聊聊这个名字，“栈”意味着什么。为此，我们需要一个稍微更复杂一点的例子：

```
fn foo() {
    let y = 5;
    let z = 100;
}

fn main() {
    let x = 42;

    foo();
}
```

这个程序总共有3个变量： `foo()` 中两个， `main()` 中一个。就像之前一样，当 `main()` 被调用时，在它的栈帧中被分配了一个单独的整型。不过在我们展示当 `foo()` 被调用后发生了什么之前，我们需要想象一下内存中发生了什么。你的操作系统为你的程序提供了一个非常简单内存视图：一个巨大的地址列表。从 `0` 到一个很大的数字，代表你的电脑有多少内存。例如，如果你有 1GB 的内存，你的地址从 `0` 到 `1,073,741,824` 。这个数字来自 2^{30}，1GB 的字节数。gigabyte

这个内存有点像一个巨型数组：地址从 `0` 开始一直增大到最终的数字。所以这是一个我们第一个栈帧的图表：

地址	名称	值
0	x	42

我们有位于地址 `0` 的 `x` ，它的值是 `42` 。

当 `foo()` 被调用，一个新的栈帧被分配：

地址	名称	值
2	z	100
1	y	5
0	x	42

因为 `0` 被第一个帧占有， `1` 和 `2` 被用于 `foo()` 的栈帧。随着我们调用更多函数，它往上增长。

这有一些我们不得不注意的重要的内容。数字 `0` ， `1` 和 `2` 都仅仅用于说明目的，并且与编译器会实际使用的具体数字没有关系。特别的，现实中这一系列的地址将会被一定数量的用于分隔地址的字节分隔开，并且这些分隔的字节可能甚至会超过被存储的值的大小。

在 `foo()` 结束后，它的帧被释放：

地址	名称	值
0	x	42

接着，在 `main()` 之后，就连最后一个值也木有了。简单明了！

它被叫做“栈”因为它就像一叠餐盘一样工作：最先放进去的盘子是最后一个你能取出来的。为此栈有时被叫做“后进先出队列”，因为你放入栈的最后值是第一个你能取出来的值。

让我们试试第三个更深入的例子：

```
fn italic() {
    let i = 6;
}

fn bold() {
    let a = 5;
    let b = 100;
    let c = 1;

    italic();
}

fn main() {
    let x = 42;

    bold();
}
```

好的，第一步，我们调用 `main()` ：

地址	名称	值
0	x	42

接下来， `main()` 调用 `bold()` ：

地址	名称	值
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

接着 `bold()` 调用 `italic()` ：

地址	名称	值
4	*i*	*6*
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

噢！我们的栈变得很高了。

在 `italic()` 结束后，它的帧被释放，只留下 `bold()` 和 `main()` ：

地址	名称	值
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

然后接着 `bold()` 结束，只剩下 `main()` 的了：

地址	名称	值
0	x	42

接下来我们完事了。找到了窍门了吗？这就像堆盘子：你在顶部增加，从顶部取走。

堆

目前为止，它能出色的工作，不过并非所有事情都能这么运作。有时，你需要在不同函数间传递一些内存，或者让它活过一次函数执行。为此，我们可以使用堆。

在 Rust 中，你可以使用 `Box<T>` 类型在堆上分配内存。这是一个例子：

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

这是当 `main()` 被调用时内存中发生了什么：

地址	名称	值
1	y	42
0	x	??????

我们在栈上分配了两个变量的空间。 `y` 是 `42` ，一如既往，不过 `x` 怎么样呢？好吧， `x` 是一个 `Box<i32>` ，而装箱在堆上分配内存。装箱的实际值是一个带有指向“堆”指针的结构。当我们开始执行这个函数，然后 `Box::new()` 被调用，它在堆上分配了一些内存，并把 `5` 放在这。现在内存看起来像这样：

地址	名称	值
$(2^{30}) - 1$		5
...	...	...
1	y	42
0	x	→ $(2^{30}) - 1$

在我们假设的带有 1GB 内存（RAM）的电脑上我们有 $(2^{30}) - 1$ 个地址。并且因为我们的栈从 `0` 开始增长，分配内存的最简单的位置是内存的另一头。所以我们第一个值位于内存的最顶端。而在 `x` 的结构的值有一个裸指针指向我们在堆上分配的位置，所以 `x` 的值是 $(2^{30}) - 1$，我们请求的内存位置。

我们还没有过多的讨论在这个上下文中分配和释放内存具体意味着什么。深入非常底层的细节超出了这个教程的范围，不过需重要指出的是这里的堆不仅仅就是一个相反方向增长的栈。在本书的后面我们会有一个例子，不过因为堆可以以任意顺序分配和释放，它最终会产生“洞”。这是一个已经运行了一段时间的程序的内存图表：

地址	名称	值
$(2^{30}) - 1$		5
$(2^{30}) - 2$		
$(2^{30}) - 3$		
$(2^{30}) - 4$		42
...	...	...
2	z	→ $(2^{30}) - 4$
1	y	42
0	x	→ $(2^{30}) - 1$

在这个例子中，我们在堆上分配了4个东西，不过释放了它们中的两个。在 $(2^{30}) - 1$ 和 $(2^{30}) - 4$ 之间有一个目前并没有被使用的断片（gap）。如何和为什么这会发生的具体细节依赖你用来管理堆的何种策略。不同的程序可以使用不同的“内存分配器”，它们是为你管理（内存）的库。Rust 程序为此选择了 使用了jemalloc。

不管怎么说，回到我们的例子。因为这些内存在堆上，它可以比分配装箱的函数活的更久。然而在这个例子中，它并不如此。移动当函数结束，我们需要为 `main()` 释放栈帧。然而 `Box<T>` 有一些玄机：Drop。 `Box` 的 `Drop` 实现释放了当它创建时分配的内存。很好！所以当 `x` 消失时，它首先释放了分配在堆上的内存：

地址	名称	值
1	y	42
0	x	??????

接着栈帧消失，释放所有的内存。

参数和借用

我们有了一些关于栈和堆运行的基础例子，不过函数参数和借用又怎么样呢？这是一个小的 Rust 程序：

```
fn foo(i: &i32) {
    let z = 42;
}

fn main() {
    let x = 5;
    let y = &x;

    foo(y);
}
```

当我们进入 `main()` ，内存看起来像这样：

地址	名称	值
1	y	→ 0
0	x	5

`x` 是一个普通的 `5` ，而 `y` 是一个指向 `x` 的引用。所以它的值是 `x` 的所在内存位置，它在这是 `0` 。

那么当我们调用 `foo()` ，传递 `y` 作为一个参数会怎么样呢？

地址	名称	值
3	z	42
2	i	→ 0
1	y	→ 0
0	x	5

栈帧不再仅仅是本地绑定了，它也有参数。所以在这里，我们需要有 `i` 参数，和 `z` ，我们本地的变量绑定。 `i` 是参数 `y` 的一个拷贝。因为 `y` 的值是 `0` ， `i` 也是。

为什么要借用一个变量的一个原因是不需要分配任何内存：一个引用的值仅仅是一个内存位置的指针。如果我们溢出任何底层内存，事情就不能这么顺利工作了。

一个复杂的例子

好的，让我们一步一步过一遍这个复杂程序：

```
fn foo(x: &i32) {
    let y = 10;
    let z = &y;

    baz(z);
    bar(x, z);
}

fn bar(a: &i32, b: &i32) {
    let c = 5;
    let d = Box::new(5);
    let e = &d;

    baz(e);
}

fn baz(f: &i32) {
    let g = 100;
}

fn main() {
    let h = 3;
    let i = Box::new(20);
    let j = &h;

    foo(j);
}
```

首先，我们调用 `main()` ：

地址	名称	值
$(2^{30}) - 1$		20
...	...	...
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

我们为 `j` ， `i` 和 `h` 分配内存。 `i` 在堆上，所以这里我们有一个指向它的值。

下一步，在 `main()` 的末尾， `foo()` 被调用：

地址	名称	值
$(2^{30}) - 1$		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

为 `x` ， `y` 和 `z` 分配了空间。参数 `x` 和 `j` 有相同的值，因为这是我们传递给它的。它是一个指向 `0` 地址的指针，因为 `j` 指向 `h` 。

接着， `foo()` 调用 `baz()` ，传递 `z` ：

地址	名称	值
(2^{30}) - 1		20
...	...	...
7	g	100
6	f	→ 4
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

我们为 `f` 和 `g` 分配了内存。`baz()` 非常短，所以当它结束时，我们移除了它的栈帧：

地址	名称	值
(2^{30}) - 1		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ (2^{30}) - 1
0	h	3

接下来，`foo()` 调用 `bar()` 并传递 `x` 和 `z` ：

地址	名称	值
$(2^{30}) - 1$		20
$(2^{30}) - 2$		5
...	...	...
10	e	→ 9
9	d	→ $(2^{30}) - 2$
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

我们最终在堆上分配了另一个值，所以我们必须从 $(2^{30}) - 1$ 减一。它比直接写 `1,073,741,822` 更简单。在任何情况下，我们通常用这个值。

在 `bar()` 的末尾，它调用了 `baz()` ：

地址	名称	值
$(2^{30}) - 1$		20
$(2^{30}) - 2$		5
...	...	...
12	g	100
11	f	→ $(2^{30}) - 2$
10	e	→ 9
9	d	→ $(2^{30}) - 2$
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

这样，我们就到达最深的位置！噢！恭喜你一路跟了过来。

在 `baz()` 结束后，我们移除了 `f` 和 `g` ：

地址	名称	值
$(2^{30}) - 1$		20
$(2^{30}) - 2$		5
...	...	...
10	e	→ 9
9	d	→ $(2^{30}) - 2$
8	c	5
7	b	→ 4
6	a	→ 0
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

接下来，我们从 `bar()` 返回。在这里 `d` 是一个 `Box<T>` ，所以它也释放了它指向的内存空间：$(2^{30}) - 2$。

地址	名称	值
$(2^{30}) - 1$		20
...	...	...
5	z	→ 4
4	y	10
3	x	→ 0
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

而在这之后， `foo()` 返回：

地址	名称	值
$(2^{30}) - 1$		20
...	...	...
2	j	→ 0
1	i	→ $(2^{30}) - 1$
0	h	3

接着，最后， `main()` ，它清理了剩下的东西。当 `i` 被 `Drop` 时，它也会清理最后的堆空间。

其它语言怎么做？

大部分语言有一个默认堆分配的垃圾回收器。这意味着每个值都是装箱的。有很多原因为什么要这么做，不过这超出了这个教程的范畴。这也有一些优化会使得这些规则不是 100% 的时间都成立。垃圾回收器用处理堆来代替依赖栈和 `Drop` 来清理内存的方式。

该用啥？（**Which to use?**）

那么如果栈是更快并更易于管理的，那么我们为啥还需要堆呢？一个大的原因是只有栈分配的话意味着你只有后进先出语义的获取存储的方法。堆分配，严格的说，更通用，允许以任意顺序从池中取出和返回存储，不过有复杂度开销。

一般来说，你应该倾向于栈分配，因此，Rust 默认栈分配。栈的后进先出模型在基本原理层面上更简单。这有两个重大的影响：运行时效率和语义影响。

运行时效率

管理栈的内存是平凡的（trivial：来自C++的概念）：机器只是增加和减少一个单独的值，所谓的“栈指针”。管理堆的内存是不平凡的（non-trivial）：堆分配的内存在任意时刻被释放，而且每个堆分配内存的块可以是任意大小，内存管理器通常需要更多工作来识别出需要重用的内存。

如果你想深入这个主题的更多细节，这篇论文是一个很好的介绍。

语义影响（Semantic impact）

栈分配影响 Rust 语言自身，因此还有开发者的心智模型（mental model：
）。后进先出语义驱动了 Rust 语言如何处理自动内存管理。甚至是一个单独所有权堆分配的装箱的释放也可以被基于栈的后进先出语义驱动，就像本章中的讨论一样。非后进先出语义的灵活性（也就是说：表现力）意味着大体上讲编译器不能在编译时自动推断出哪些内存应该被释放；它不得不依赖动态协议，可能来自于语言之外，来驱动释放（例如， `Rc<T>` 和 `Arc<T>` 中用到了引用计数）。

当考虑到极端情况，堆分配的增加的表现力带来了要么是显著的运行时支持（例如，以垃圾回收器的形式）要么是显著的程序猿努力（以必需进行 Rust 编译器并未提供的验证的显式的内存管理调用的形式）的开销。

> gigabyte. `Gigabyte` 可以表示两个意思：10^9 或 2^{30}。SI（国际单位制）标准解释为“gigabyte”是 10^9， 而“gibibyte”是 2^{30}。然而，很少有人用这个术语，而依赖语境上下文来区别。这里我们遵循传统。↩

> 移动. 我们可以通过转移所有权来让内存活的更久，这有时叫做“移出箱子”。我们将在后面涉及更复杂的例子。↩

测试

> testing.md
> commit d7376702d21ad31bdbea2f41bc701d7cac837183
>
> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
>
> Edsger W. Dijkstra, "The Humble Programmer" (1972)
>
> 软件测试是证明 bug 存在的有效方法，而证明它们不存在时则显得令人绝望的不足。
>
> Edsger W. Dijkstra，【谦卑的程序员】（1972）

让我们讨论一下如何测试 Rust 代码。在这里我们不会讨论什么是测试 Rust 代码的正确方法。有很多关于写测试好坏方法的流派。所有的这些途径都使用相同的基本工具，所以我们会向你展示他们的语法。

`test` 属性（The test attribute）

简单的说，测试是一个标记为 `test` 属性的函数。让我们用 Cargo 来创建一个叫 `adder` 的项目：

```
$ cargo new adder
$ cd adder
```

在你创建一个新项目时 Cargo 会自动生成一个简单的测试。下面是 `src/lib.rs` 的内容：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
    }
}
```

现在暂时去掉 `mod` 那部分，只关注函数：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
#[test]
fn it_works() {
}
```

注意这个 `#[test]` 。这个属性表明这是一个测试函数。它现在没有函数体。它肯定能编译通过！让我们用 `cargo test` 运行测试：

```
$ cargo test
   Compiling adder v0.1.0 (file:///home/you/projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.15 s
ecs
     Running target/debug/deps/adder-941f01916ca4a642

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Cargo 编译和运行了我们的测试。这里有两部分输出：一个是我们写的测试，另一个是文档测试。我们稍后再讨论这些。现在，看看这行：

```
test tests::it_works ... ok
```

注意那个 `tests::it_works` 。这是我们函数的名字：

```
# fn main() {
fn it_works() {
}
# }
```

然后我们有一个总结行：

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

那么为啥我们这个啥都没干的测试通过了呢？任何没有 `panic!` 的测试通过，`panic!` 的测试失败。让我们的测试失败：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!` 是 Rust 提供的一个宏，它接受一个参数：如果参数是 `true` ，啥也不会发生。如果参数是 `false` ，它会 `panic!` 。让我们再次运行我们的测试：

```
$ cargo test
   Compiling adder v0.1.0 (file:///home/you/projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.17 s
ecs
     Running target/debug/deps/adder-941f01916ca4a642

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
        thread 'it_works' panicked at 'assertion failed: false',
 src/lib.rs:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.


failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

error: test failed
```

Rust 指出我们的测试失败了：

```
test it_works ... FAILED
```

这反映在了总结行上：

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

我们也得到了一个非 0 的状态码.我们在 OS X和 Linux 中使用 `$?` ：

```
$ echo $?
101
```

在 Windows 中，如果你使用 `cmd` ：

```
> echo %ERRORLEVEL%
```

而如果你使用 PowerShell：

```
> echo $LASTEXITCODE # the code itself
> echo $? # a boolean, fail or succeed
```

这在你想把 `cargo test` 集成进其它工具时是非常有用。

我们可以使用另一个属性反转我们的失败的测试： `should_panic` ：

```
# // The next line exists to trick play.rust-lang.org into running our code as a
# // test:
# // fn main
#
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

现在即使我们 `panic!` 了测试也会通过，并且如果我们的测试通过了则会失败。让我试一下：

```
$ cargo test
   Compiling adder v0.1.0 (file:///home/you/projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.17 s
ecs
     Running target/debug/deps/adder-941f01916ca4a642

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust 提供了另一个宏，`assert_eq!` 用来比较两个参数：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
#[test]
#[should_panic]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

那个测试通过了吗？因为那个 `should_panic` 属性，它通过了：

```
$ cargo test
   Compiling adder v0.1.0 (file:///home/you/projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.21 s
ecs
     Running target/debug/deps/adder-941f01916ca4a642

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`should_panic` 测试是脆弱的，因为很难保证测试是否会因什么不可预测原因并未失败。为了解决这个问题， `should_panic` 属性可以添加一个可选的 `expected` 参数。这个参数可以确保失败信息中包含我们提供的文字。下面是我们例子的一个更安全的版本：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

这就是全部的基础内容！让我们写一个“真实”的测试：

```
# // The next line exists to trick play.rust-lang.org into running our code as a
# // test:
# // fn main
#
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

`assert_eq!` 是非常常见的；用已知的参数调用一些函数然后与期望的输出进行比较。

`ignore` 属性

有时一些特定的测试可能非常耗时。这时可以通过 `ignore` 属性来默认禁用：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}

#[test]
#[ignore]
fn expensive_test() {
    // Code that takes an hour to run...
}
```

现在我们运行测试并发现 `it_works` 被执行了，而 `expensive_test` 没有

```
$ cargo test
   Compiling adder v0.1.0 (file:///home/you/projects/adder)
    Finished debug [unoptimized + debuginfo] target(s) in 0.20 s
ecs
     Running target/debug/deps/adder-941f01916ca4a642

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

耗时的测试可以通过调用 `cargo test -- --ignored` 来执行：

```
$ cargo test -- --ignored
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 se
cs
     Running target/debug/deps/adder-941f01916ca4a642

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`--ignored` 参数是 test 程序的参数，而不是 Cargo 的，这也是为什么命令是 `cargo test -- --ignored` 。

`tests` 模块

然而以这样的方式来实现我们的测试的例子并不是地道的做法：它缺少 `tests` 模块。你可能注意到了这个测试模块在最初用 `cargo new` 生成时还在代码中存在，不过在我们最后一个例子中消失了。让我们解释一下。

一个比较惯用的做法应该是如下的：

```
# // The next line exists to trick play.rust-lang.org into running our code as a
# // test:
# // fn main
#
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::add_two;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

这里产生了一些变化。第一个变化是引入了一个 `cfg` 属性的 `mod tests` 。这个模块允许我们把所有测试集中到一起，并且需要的话还可以定义辅助函数，它们不会成为我们包装箱的一部分。 `cfg` 属性只会在我们尝试去运行测试时才会编译测试代码。这样可以节省编译时间，并且也确保我们的测试代码完全不会出现在我们的正式构建中。

第二个变化是 `use` 声明。因为我们在一个内部模块中，我们需要把我们要测试的函数导入到当前空间中。如果你有一个大型模块的话这会非常烦人，所以这里有经常使用一个 `glob` 功能。让我们修改我们的 `src/lib.rs` 来使用这个：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

注意 `use` 行的变化。现在运行我们的测试：

```
$ cargo test
    Updating registry `https://github.com/rust-lang/crates.io-in
dex`
   Compiling adder v0.1.0 (file:///home/you/projects/adder)
     Running target/debug/deps/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

它能工作了！

目前的习惯是使用 `test` 模块来存放你的“单元测试”。任何只是测试一小部分功能的测试理应放在这里。那么“集成测试”怎么办呢？我们有 `tests` 目录来处理这些。

`tests` 目录

每一个 `tests/*.rs` 文件都被当作一个独立的 crate。因此，为了进行集成测试，让我们创建一个 `tests` 目录，然后放一个 `tests/integration_test.rs` 文件进去，输入如下内容：

```
# // The next line exists to trick play.rust-lang.org into running our code as a
# // test:
# // fn main
#
# // Sadly, this code will not work in play.rust-lang.org, because we have no
# // crate adder to import. You'll need to try this part on your own machine.
extern crate adder;

#[test]
fn it_works() {
    assert_eq!(4, adder::add_two(2));
}
```

这看起来与我们刚才的测试很像，不过有些许的不同。我们现在有一行 `extern crate adder` 在开头。这是因为在 `tests` 目录中的每个测试（文件）是一个完全不同的 crate，所以我们需要导入我们的库。这也是为什么 `tests` 是一个写集成测试的好地方：它们就像其它程序一样使用我们的库。

让我们运行一下：

```
$ cargo test
   Compiling adder v0.1.0 (file:///home/you/projects/adder)
     Running target/debug/deps/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/integration_test-68064b69521c828a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

现在我们有了三个部分：我们之前的两个测试，然后还有我们新添加的。

Cargo（不？）会忽略 `tests/` 目录的子目录的文件。因此在集成测试中共享模块是可能的。例如 `tests/common/mod.rs` 并不会被 Cargo 单独编译并可以被任何包含 `mod common` 的测试（文件）引用。

这就是 `tests` 目录的全部内容。它不需要 `test` 模块因为它整个就是关于测试的。

让我们最后看看第三部分：文档测试。

文档测试

没有什么是比带有例子的文档更好的了。当然也没有什么比不能工作的例子更糟的，因为文档完成之后代码已经被改写。为此，Rust支持自动运行你文档中的例子（注意：这只在库 crate中有用，而在二进制 crate 中没用）。这是一个完整的有例

子的 `src/lib.rs` ：

```
# // The next line exists to trick play.rust-lang.org into runni
ng our code as a
# // test:
# // fn main
#
//! The `adder` crate provides functions that add numbers to oth
er numbers.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

注意模块级的文档以 `//!` 开头然后函数级的文档以 `///` 开头。Rust文档在注释中支持 Markdown 语法，所以它支持 3 个反单引号代码块语法。想上面例子那样，加入一个 `# Examples` 部分被认为是一个惯例。

让我们再次运行测试：

```
$ cargo test
   Compiling adder v0.1.0. (file:///home/you/projects/adder)
     Running target/debug/deps/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/integration_test-68064b69521c828a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 2 tests
test add_two_0 ... ok
test _0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

现在我们运行了3种测试！注意文档测试的名称：`_0` 生成为模块测试，而 `add_two_0` 函数测试。如果你添加更多用例的话它们会像 `add_two_1` 这样自动加一。

我们还没有讲到所有编写文档测试的所有细节。关于更多，请看文档章节。

测试与并发

特别需要注意的是测试使用线程来并发的运行。为此需要注意测试之间不能相互依

赖也不能依赖任何共享状态。“共享状态”可以包括运行环境，例如当前工作目录（cwd），或者环境变量。

如果这样做有问题控制这些并发也是可能的，要么设置环境变量 `RUST_TEST_THREADS`，或者向测试传递 `--test-threads` 用来比较两个参数：

```
$ RUST_TEST_THREADS=1 cargo test   # Run tests with no concurren
cy
...
$ cargo test -- --test-threads=1   # Same as above
...
```

测试输出

默认 Rust 测试标准库捕获并将输出丢弃到标准输出/错误中。例如来自 `println!()` 的输出。这也可以通过环境变量或者参数来控制：

```
$ RUST_TEST_NOCAPTURE=1 cargo test   # Preserve stdout/stderr
...
$ cargo test -- --nocapture          # Same as above
...
```

然而一个避免输出被捕获的更好的方式是采用日志而不是使用原始的输出。Rust 有一个 standard logging API，它提供了一个多种日志实现的前端。这可以被用来与默认的 env_logger 相结合，以一种可以在运行时控制的方式输出任何调试信息。

条件编译

> conditional-compilation.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

Rust 有一个特殊的属性，`#[cfg]` ，它允许你基于一个传递给编译器的标记编译代码。它有两种形式：

```
#[cfg(foo)]
# fn foo() {}

#[cfg(bar = "baz")]
# fn bar() {}
```

它还有一些帮助选项：

```
#[cfg(any(unix, windows))]
# fn foo() {}

#[cfg(all(unix, target_pointer_width = "32"))]
# fn bar() {}

#[cfg(not(foo))]
# fn not_foo() {}
```

这些选项可以任意嵌套：

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
# fn foo() {}
```

至于如何启用和禁用这些开关，如果你使用 Cargo 的话，它们可以在你 `Cargo.toml` 中的 `[features]` 部分设置：

```
[features]
# no features by default
default = []

# Add feature "foo" here, then you can use it.
# Our "foo" feature depends on nothing else.
foo = []
```

当你这么做的时候，Cargo传递给 `rustc` 一个标记：

```
--cfg feature="${feature_name}"
```

这些 `cfg` 标记集合会决定哪些功能被启用，并且因此，哪些代码会被编译。让我们看看这些代码：

```
#[cfg(feature = "foo")]
mod foo {
}
```

如果你用 `cargo build --features "foo"` 编译，他会向 `rustc` 传递 `--cfg feature="foo"` 标记，并且输出中将会包含 `mod foo` 。如果我们使用常规的 `cargo build` 编译，则不会传递额外的标记，因此，（输出）不会存在 `foo` 模块。

cfg_attr

你也可以通过一个基于 `cfg` 变量的 `cfg_attr` 来设置另一个属性：

```
#[cfg_attr(a, b)]
# fn foo() {}
```

如果 `a` 通过 `cfg` 属性设置了的话这与 `#[b]` 相同，否则不起作用。

cfg!

`cfg!` 语法扩展也让你可以在你的代码中使用这类标记：

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

这会在编译时被替换为一个 `true` 或 `false` ，依配置设定而定。

文档

> documentation.md
> commit d7376702d21ad31bdbea2f41bc701d7cac837183

在任何软件项目中，文档都是重要的部分，其同样在 Rust 中是头等重要的。让我们讨论下 Rust 提供给我们的编写项目文档的工具。

关于 `rustdoc`

Rust发行版中包含了一个工具， `rustdoc` ，它可以生成文档。 `rustdoc` 也可以在 Cargo 中通过 `cargo doc` 使用。

文档可以使用两种方法生成：从源代码，或者从单独的 Markdown 文件。

文档化源代码

文档化 Rust 项目的主要方法是在源代码中添加注释。你可以使用文档注释以实现此目的：

```
/// Constructs a new `Rc<T>`.
///
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
pub fn new(value: T) -> Rc<T> {
    // Implementation goes here.
}
```

这段代码产生像这样的文档。我忽略了函数的实现，而在此处留下了一个常规注释。

这个注释首先需要注意的地方是它使用了 `///`，而不是 `//`。三条斜线表明这是文档注释。

文档注释用 Markdown 语法编写。

Rust 会跟踪这些注释，并在生成文档时使用它们。这在文档化像枚举这样的结构时很重要：

```
/// The `Option` type. See [the module level documentation](../)
 for more.
enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

以上代码可以工作，但这个就不行了：

```
/// The `Option` type. See [the module level documentation](../)
 for more.
enum Option<T> {
    None, /// No value
    Some(T), /// Some value `T`
}
```

你会得到一个错误：

```
hello.rs:4:1: 4:2 error: expected ident, found `}`
hello.rs:4 }
           ^
```

这个不幸的错误是有道理的；文档注释适用于它后面的内容，而在最后的注释后面没有任何内容。

编写文档注释

不管怎样，让我们来详细了解一下注释的每一部分：

```
/// Constructs a new `Rc<T>`.
# fn foo() {}
```

文档注释的第一行应该是其功能的一个简短摘要。一句话。唯基础。高层次。

```
///
/// Other details about constructing `Rc<T>`s, maybe describing
complicated
/// semantics, maybe additional options, all kinds of stuff.
///
# fn foo() {}
```

我们原始的例子只有一行摘要，不过如果有更多东西要写，我们可以新开一个新的段落增加更多解释。

特殊部分

下面，是特殊部分。它由一个标头指示，`#`。有四种经常使用的标头。到目前为止，它们不是特殊语法，只是传统。

```
/// # Panics
# fn foo() {}
```

Rust 中不可恢复的函数滥用（如程序错误）通常用恐慌（panics）表示，它至少也要杀死整个当前线程。如果你的函数有这样一个非凡的契约，需要识别或者强制引发恐慌，记录文档是非常重要的。

```
/// # Errors
# fn foo() {}
```

如果你的函数或方法返回 `Result<T, E>`，那么描述何种情况下它会返回 `Err(E)` 是件值得做的好事。这并不如 `Panics` 重要，因为失败被编码进了类型系统，不过仍旧值得一做。

```
/// # Safety
# fn foo() {}
```

如果你的函数是`unsafe`的，你应该解释调用者需负责维护哪些不可变量。

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
# fn foo() {}
```

第四个，`Examples`。包含一个或多个使用你的函数或方法的例子，这样你的用户会为此感谢你的。这些例子写在代码块注释中（我们稍后会讨论到），并且可以有不止一个段落：

```
/// # Examples
///
/// Simple `&str` patterns:
///
/// ```
/// let v: Vec<&str> = "Mary had a little lamb".split(' ').colle
ct();
/// assert_eq!(v, vec!["Mary", "had", "a", "little", "lamb"]);
/// ```
///
/// More complex patterns with a lambda:
///
/// ```
/// let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeri
c()).collect();
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ```
# fn foo() {}
```

让我们聊聊这些代码块的细节。

代码块注释

要在注释中写Rust代码，使用三个重音号：

```
/// ```
/// println!("Hello, world");
/// ```
# fn foo() {}
```

如果你要写一些非Rust的代码，你可以加上注解：

```
/// ```c
/// printf("Hello, world\n");
/// ```
# fn foo() {}
```

这会根据你选择的语言高亮代码。如果你只是想展示普通文本，选择 `text` 。

选择正确的注释是重要的，因为 `rustdoc` 用一种有意思的方法使用它：在库crate中它可以用来实际测试你的示例代码，使之不至于过时。如果你写了C代码但是忽略了注解， `rustdoc` 会认为它是Rust代码，会在你生成文档时提示。

用作测试的文档

讨论下我们的样本示例文档：

```
/// ```
/// println!("Hello, world");
/// ```
# fn foo() {}
```

你会注意到这里并不需要 `fn main()` 或者别的东西。 `rustdoc` 会自动加一个 `main()` 来包裹你的代码，用试探法来把它放到正确的位置。例如：

```
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
# fn foo() {}
```

这是最终的测试：

```
fn main() {
    use std::rc::Rc;
    let five = Rc::new(5);
}
```

下面是 `rustdoc` 预处理示例的完整算法：

1. 任何 `#![foo]` 开头的属性会被完整的作为包装箱属性
2. 一些通用的 `allow` 属性被插入，包括 `unused_variables`、`unused_assignments`、`unused_mut`、`unused_attributes` 和 `dead_code`。小的例子经常触发这些lint检查
3. 如果例子并未包含 `extern crate`，那么 `extern crate <mycrate>;` 被插入（注意缺失了 `#[macro_use]`）
4. 最后，如果例子不包含 `fn main`，剩下的文本将被包装到 `fn main() { your_code }` 中

有时，这是不够的。例如，我们已经考虑到了所有 `///` 开头的代码样例了吗？普通文本：

```
/// Some documentation.
# fn foo() {}
```

与它的输出看起来有些不同：

```
/// Some documentation.
# fn foo() {}
```

是的，你猜对了：你写的以 `#` 开头的行会在输出中被隐藏，不过会在编译你的代码时被使用。你可以利用这一点。在这个例子中，文档注释需要适用于一些函数，所以我只想向你展示文档注释，我需要在下面增加一些函数定义。同时，这只是用来满足编译器的，所以省略它会使得例子看起来更清楚。你可以使用这个技巧来详细的解释较长的例子，同时保留你文档的可测试行。

例如，想象一下我们想要为如下代码写文档：

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

最终我们可能想要文档变成这样：

> 首先，我们把 `x` 设置为 `5` ：
>
> ```
> let x = 5;
> # let y = 6;
> # println!("{}", x + y);
> ```
>
> 接着，我们把 `y` 设置为 `6` ：
>
> ```
> # let x = 5;
> let y = 6;
> # println!("{}", x + y);
> ```
>
> 最后，我们打印 `x` 和 `y` 的和：
>
> ```
> # let x = 5;
> # let y = 6;
> println!("{}", x + y);
> ```

为了让每个代码块可以执行，我们想要每个代码块都有整个程序，不过我们并不想读者每回都看到所有的行。这是我们的源代码：

````
首先，我们把`x`设置为`5`：

```text
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

接着，我们把`y`设置为`6`：

```text
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

最后，我们打印`x`和`y`的和：

```text
# let x = 5;
# let y = 6;
println!("{}", x + y);
```
````

通过重复例子的所有部分，你可以确保你的例子仍能编译，同时只显示与你解释相关的部分。

文档化宏

下面是一个宏的文档例子：

```
/// Panic with a given message unless an expression evaluates to
 true.
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, “Math is broken.”);
/// # }
/// ```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, “I’m broken.”);
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { p
anic!($($rest),+); } });
}
# fn main() {}
```

你会注意到3个地方：我们需要添加我们自己的 `extern crate` 行，这样我们可以添加 `#[macro_use]` 属性。第二，我们也需要添加我们自己的 `main()` （为了上面讨论过的原因）。最后，用 `#` 机智的注释掉这两个代码，这样它们不会出现在输出中。

另一个 `#` 好用的情况是当你想要忽略错误处理的时候。例如你想要如下情况。

```
/// use std::io;
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
```

问题是 `try!` 返回一个 `Result<T, E>` 而测试函数并不返回任何值所以这会产生一个类型不匹配错误。

```
/// A doc test using try!
///
/// ```
/// use std::io;
/// # fn foo() -> io::Result<()> {
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
/// # Ok(())
/// # }
/// ```
# fn foo() {}
```

你可以将代码放进函数里来解决这个问题。在运行文档测试时它捕获并返回 `Result<T, E>` 。这种模式不时出现在标准库中。

运行文档测试

要运行测试，要么

```
$ rustdoc --test path/to/my/crate/root.rs
# or（或者）
$ cargo test
```

对了， `cargo test` 也会测试内嵌的文档。然而， **`cargo test`** 将不会测试二进制 **crate**，只测试库 **crate**。这是由于 `rustdoc` 的运行机制：它链接要测试的库，不过对于一个二进制文件，木有什么好链接的。

这还有一些注释有利于帮助 `rustdoc` 在测试你的代码时正常工作：

```
/// ```ignore
/// fn foo() {
/// ```
# fn foo() {}
```

`ignore` 指令告诉Rust忽略你的代码。这几乎不会是你想要的，因为这是最不受支持的。相反，如果不是代码的话考虑注释为 `text` ，或者使用 `#` 来形成一个可运行但只显示你关心部分的例子。

```
/// ```should_panic
/// assert!(false);
/// ```
# fn foo() {}
```

`should_panic` 告诉 `rustdoc` 这段代码应该正确编译，但是作为一个测试则不能通过。

```
/// ```no_run
/// loop {
///     println!("Hello, world");
/// }
/// ```
# fn foo() {}
```

`no_run` 属性会编译你的代码，但是不运行它。这对像如“如何开始一个网络服务”这样的例子很重要，你会希望确保它能够编译，不过它可能会在一个没有网络连接的测试环境中运行。

文档化模块

Rust 有另一种文档注释， `//!` 。这种注释并不文档化接下来的内容，而是包围它的内容。换句话说：

```
mod foo {
    //! This is documentation for the `foo` module.
    //!
    //! # Examples

    // ...
}
```

这是你会看到 `//!` 最常见的用法：作为模块文档。如果你在 `foo.rs` 中有一个模块，打开它你常常会看到这些：

```
//! A module for using `foo`s.
//!
//! The `foo` module contains a lot of useful functionality blah
 blah blah...
```

Crate 文档

Crate 文档可以通过在 crate 根文件，也就是 `lib.rs` ，的开头放置文档内注释（ `//!` ）来编写：

```
//! This is documentation for the `foo` crate.
//!
//! The foo crate is meant to be used for bar.
```

文档注释风格

查看RFC 505以了解文档风格和格式的惯例。

其它文档

所有这些行为都能在非 Rust 代码文件中工作。因为注释是用 Markdown 编写的，它们通常是 `.md` 文件。

当你在 Markdown 文件中写文档时，你并不需要加上注释前缀。例如：

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
# fn foo() {}
```

在一个 Markdown 文件中，就是：

````
# Examples

```
use std::rc::Rc;

let five = Rc::new(5);
```
````

不过文档写在 Markdown 文件中要加一点：Markdown文件需要有一个像这样的标题：

```
% The title

This is the example documentation.
```

`%` 行需要放在文件的第一行。

`doc` 属性

在更底层，文档注释是文档属性的语法糖：

```
/// this
# fn foo() {}

#[doc="this"]
# fn bar() {}
```

跟下面这个是类似的：

```
//! this

#![doc="this"]
```

写文档时你不会经常看见这些属性，不过当你要改变一些选项，或者写一个宏的时候比较有用。

重导出（Re-exports）

`rustdoc` 对公有重导出部分会在两个地方都显示文档：

```
extern crate foo;

pub use foo::bar;
```

这既会为 `bar` 在 `foo` 包装箱中生成文档，也会在你的包装箱中生成文档。它会在两个地方使用相同的内容。

这种行为可以通过 `no_inline` 来阻止：

```
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

缺失文档

有时你想要确保你项目中每一个公开的项都有文档，特别是当你编写一个库的时候。Rust 允许你为此产生警告或错误，当一个项缺少文档时。为了生成警告，你需要使用 `warn` ：

```
#![warn(missing_docs)]
```

而为了生成错误你需要使用 `deny` ：

```
#![deny(missing_docs)]
```

有时你想要显式的禁用警告/错误来让一些项没有文档。这可以使用 `allow` 来搞定：

```
#[allow(missing_docs)]
struct Undocumented;
```

你可能甚至希望在文档中完全隐藏某些项：

```
#[doc(hidden)]
struct Hidden;
```

控制 HTML

你可以通过 `#![doc]` 属性控制 `rustdoc` 生成的 HTML 文档的几个地方：

```
#![doc(html_logo_url = "https://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
       html_favicon_url = "https://www.rust-lang.org/favicon.ico",
       html_root_url = "https://doc.rust-lang.org/")]
```

这里设置了一些不同的选项，带有一个logo，一个网站图标，和一个根URL。

配置文档测试

你也可以通过 `#![doc(test(..))]` 属性来配置 `rustdoc` 测试你文档示例的方式。

```
#![doc(test(attr(allow(unused_variables), deny(warnings))))]
```

这允许示例中存在未使用的变量，但其他 lint 警告抛出仍会使测试失败。

生成选项

`rustdoc` 也提供了一些其他命令行选项，以便进一步定制：

- `--html-in-header FILE` ：在 `<head>...</head>` 部分的末尾加

上 `FILE` 内容

- `--html-before-content FILE`：在 `<body>` 之后，在渲染内容之前加上 `FILE` 内容
- `--html-after-content FILE`：在所有渲染内容之后加上 `FILE` 内容

安全事项

文档注释中的 Markdown 会被不加以处理地放置于最终的网页中。注意原始的 HTML 文本：

```
/// <script>alert(document.cookie)</script>
# fn foo() {}
```

迭代器

> iterators.md
>
> commit 27602e47dbd24b1ce273e54a82eca32cc231794d

让我们讨论一下循环。

还记得 Rust 的 `for` 循环吗？这是一个例子：

```
for x in 0..10 {
    println!("{}", x);
}
```

现在我们更加了解 Rust 了，我们可以谈谈这里的具体细节了。这个范围（ `0..10` ）是“迭代器”。我们可以重复调用迭代器的 `.next()` 方法，然后它会给我们一个数据序列。

（另外，像 `0..10` 带有两个点号的 range 是包含左边（从 0 开始）但不包含右边的值（到 9 为止）。一个数学家会这么写“[0, 10)”。为了得到一个一个一直到 10 的 range 你可以写成 `0...10` 。）

就像这样：

```
let mut range = 0..10;

loop {
    match range.next() {
        Some(x) => {
            println!("{}", x);
        },
        None => { break }
    }
}
```

我们创建了一个 `range` 的可变绑定，它是我们的迭代器。我们接着 `loop` ，它包含一个 `match` 。 `match` 用来匹配 `range.next()` 的结果，它给我们迭代器的下一个值。 `next` 返回一个 `Option<i32>` ，在这个例子中，如果有值,它会返

回 `Some(i32)` 然后当我们循环完毕,就会返回 `None` 。如果我们得到 `Some(i32)` ，我们就会打印它，如果我们得到 `None` ，我们 `break` 出循环。

这个代码例子基本上和我们的 `loop` 版本一样。 `for` 只是 `loop` / `match` / `break` 结构的简便写法。

然而， `for` 循环并不是唯一使用迭代器的结构。编写你自己的迭代器涉及到实现 `Iterator` 特性。然而特性不是本章教程的涉及范围，不过 Rust 提供了一系列的有用的迭代器帮助我们完成各种任务。但首先注意下范围 的一些局限性。

范围 非常原始，我们通常可以用更好的替代方案。考虑下面的 Rust 反模式：用范围 来模拟 C-风格的 `for` 循环。比如你想遍历完 vector 的内容。你可能尝试这么写：

```
let nums = vec![1, 2, 3];

for i in 0..nums.len() {
    println!("{}", nums[i]);
}
```

这严格的说比使用现成的迭代器还要糟。你可以直接在 vector 上遍历。所以这么写：

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", num);
}
```

这么写有两个原因。第一，它更明确的表明了我们的意图。我们迭代整个向量，而不是先迭代向量的索引，再按索引取值。第二，这个版本也更有效率：第一个版本会进行额外的边界检查因为它使用了索引， `nums[i]` 。因为我们利用迭代器获取每个向量元素的引用，第二个例子中并没有边界检查。这在迭代器中非常常见：我们可以忽略不必要的边界检查，不过仍然知道我们是安全的。

这里还有一个细节不是 100% 清楚的就是 `println!` 是如何工作的。 `num` 是 `&i32` 类型。也就是说，它是一个 `i32` 的引用，并不是 `i32` 本身。 `println!` 为我们处理了解引用，所以我们并没有看到它。下面的代码也能工

作：

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", *num);
}
```

现在我们显式的解引用了 `num` 。为什么 `&nums` 会给我们一个引用呢？首先，因为我们显式的使用了 `&` 。其二，如果它给我们数据，我们就是它的所有者了，这会涉及到生成数据的拷贝然后返回给我们拷贝。通过引用，我们只是借用了一个数据的引用，所以仅仅是传递了一个引用，并不涉及数据的移动。

那么，既然现在我们已经明确了范围通常不是我们需要的，那么让我们来讨论下你真正需要什么。

这里涉及到大体上相关的3类事物：迭代器，迭代适配器（*iterator adapters*）和消费者（*consumers*）。下面是一些定义：

- 迭代器 给你一个值的序列
- 迭代适配器 操作迭代器，产生一个不同输出序列的新迭代器
- 消费者 操作迭代器，产生最终值的集合

让我们先看看消费者，因为我们已经见过范围这个迭代器了。

消费者（Consumers）

消费者 操作一个迭代器，返回一些值或者几种类型的值。最常见的消费者是 `collect()` 。这个代码还不能编译，不过它表明了我们的意图：

```
let one_to_one_hundred = (1..101).collect();
```

如你所见，我们在迭代器上调用了 `collect()` 。 `collect()` 从迭代器中取得尽可能多的值，然后返回结果的集合。那么为什么这不能编译呢？因为Rust不能确定你想收集什么类型的值，所以你需要让它知道。下面是一个可以编译的版本：

```
let one_to_one_hundred = (1..101).collect::<Vec<i32>>();
```

如果你还记得，`::<>` 语法允许我们给出一个类型提示，所以我们可以告诉编译器我们需要一个整型的向量。但是你并不总是需要提供完整的类型。使用 `_` 可以让你提供一个部分的提示：

```
let one_to_one_hundred = (1..101).collect::<Vec<_>>();
```

这是指“请把值收集到 `Vec<T>` ，不过自行推断 `T` 类型”。为此 `_` 有时被称为“类型占位符”。

`collect()` 是最常见的消费者，不过这还有其它的消费者。 `find()` 就是一个：

```
let greater_than_forty_two = (0..100)
                             .find(|x| *x > 42);

match greater_than_forty_two {
    Some(_) => println!("Found a match!"),
    None => println!("No match found :("),
}
```

`find` 接收一个闭包，然后处理迭代器中每个元素的引用。如果这个元素是我们要找的，那么这个闭包返回 `true` ，如果不是就返回 `false` 。因为我们可能不能找到任何元素，所以 `find` 返回 `Option` 而不是元素本身。

另一个重要的消费者是 `fold` 。他看起来像这样：

```
let sum = (1..4).fold(0, |sum, x| sum + x);
```

`fold()` 看起来像这样： `fold(base, |accumulator, element| ...)` 。它需要两个参数：第一个参数叫做基数（*base*）。第二个是一个闭包，它自己也需要两个参数：第一个叫做累计数（*accumulator*），第二个叫元素（*element*）。每次迭代，这个闭包都会被调用，返回值是下一次迭代的累计数。在我们的第一次迭代，累计数的值是基数。

好吧，这有点混乱。让我们检查一下这个迭代器中所有这些值：

基数	累计数	元素	闭包结果
0	0	1	1
0	1	2	3
0	3	3	6

我们可以使用这些参数调用 `fold()` ：

```
# (1..4)
.fold(0, |sum, x| sum + x);
```

那么， `0` 是我们的基数， `sum` 是累计数， `x` 是元素。在第一次迭代，我们设置 `sum` 为 `0` ，然后 `x` 是 `nums` 的第一个元素， `1` 。我们接着把 `sum` 和 `x` 相加，得到 `0 + 1 = 1` 。在我们第二次迭代， `sum` 成为我们的累计值，元素是数组的第二个值， `2` ， `1 + 2 = 3` ，然后它就是最后一次迭代的累计数。在这次迭代中， `x` 是最后的元素， `3` ，那么 `3 + 3 = 6` ，就是我们和的最终值。 `1 + 2 + 3 = 6` ，这就是我们的结果。

哇。最开始几次你见到 `fold` 的时候可能觉得有点奇怪，不过一旦你习惯了它，你就会在到处都用它。任何时候你有一个列表，然后你需要一个单一的结果， `fold` 就是合适的。

消费者很重要还因为另一个我们没有讨论到的迭代器的属性：惰性。让我们更多的讨论一下迭代器，你就知道为什么 `消费者` 重要了。

迭代器（Iterators）

正如我们之前说的，迭代器是一个我们可以重复调用它的 `.next()` 方法，然后它会给我们一个数据序列的结构。因为你需要调用函数，这意味着迭代器是惰性的（*lazy* ）并且不需要预先生成所有的值。例如，下面的代码并没有真正的生成 `1-99` 这些数，而是创建了一个值来代表这个序列：

```
let nums = 1..100;
```

因为我们没有用范围做任何事，它并未生成序列。让我们加上消费者：

```
let nums = (1..100).collect::<Vec<i32>>();
```

现在，`collect()` 会要求范围生成一些值，接着它会开始产生序列。

范围是你会见到的两个基本迭代器之一。另一个是 `iter()` 。`iter()` 可以把一个向量转换为一个简单的按顺序给出每个值的迭代器：

```
let nums = vec![1, 2, 3];

for num in nums.iter() {
    println!("{}", num);
}
```

这两个基本迭代器应该能胜任你的工作。还有一些高级迭代器，包括一个是无限的。

关于迭代器的介绍足够了。迭代适配器是关于迭代器最后一个要介绍的内容了。让我们开始吧！

迭代适配器（Iterator adapters）

迭代适配器（*Iterator adapters*）获取一个迭代器然后按某种方法修改它，并产生一个新的迭代器。最简单的是一个是 `map` ：

```
(1..100).map(|x| x + 1);
```

在其他迭代器上调用 `map` ，然后产生一个新的迭代器，它的每个元素引用被调用了作为参数的闭包。所以它会给我们 `2-100` 这些数字。好吧，看起来是这样。如果你编译这个例子，你会得到一个警告：

```
warning: unused result which must be used: iterator adaptors are
 lazy and
        do nothing unless consumed, #[warn(unused_must_use)] on
 by default
(1..100).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

又是惰性！那个闭包永远也不会执行。这个例子也不会打印任何数字：

```
(1..100).map(|x| println!("{}", x));
```

如果你尝试在一个迭代器上执行带有副作用的闭包，不如直接使用 `for` 。

有大量有趣的迭代适配器。 `take(n)` 会返回一个源迭代器下 `n` 个元素的新迭代器，注意这对源迭代器没有副作用。让我们试试我们之前的无限迭代器， `count()` ：

```
for i in (1..).take(5) {
    println!("{}", i);
}
```

这会打印：

```
1
2
3
4
5
```

`filter()` 是一个带有一个闭包参数的适配器。这个闭包返回 `true` 或 `false` 。 `filter()` 返回的新迭代器只包含闭包返回 `true` 的元素：

```
for i in (1..100).filter(|&x| x % 2 == 0) {
    println!("{}", i);
}
```

这会打印出 1 到 100 之间所有的偶数。（注意：不像 `map` ，传递给 `filter` 闭包传递了一个元素的引用而不是元素本身。这里定义的过滤器使用 `&x` 模式来提取整型。过滤器闭包传递的是一个引用因为它返回 `true` 或 `false` 而不是元素，所以过滤器的实现必须保持元素的所有权并传递给新创建的迭代器。）

（注意因为 `filter` 并不消费它迭代的元素，它传递每个元素的引用，所以过滤器使用 `&x` 来提取其中的整型数据。）

你可以链式的调用所有三种结构：以一个迭代器开始，适配几次，然后处理结果。看看下面的：

```
(1..)
    .filter(|&x| x % 2 == 0)
    .filter(|&x| x % 3 == 0)
    .take(5)
    .collect::<Vec<i32>>();
```

这会给你一个包含 `6` ， `12` ， `18` ， `24` 和 `30` 的向量。

这只是一个迭代器、迭代适配器和消费者如何帮助你的小尝试。有很多非常实用的迭代器，当然你也可以编写你自己的迭代器。迭代器提供了一个安全、高效的处理所有类型列表的方法。最开始它们显得比较不寻常，不过如果你玩转了它们，你就会上瘾的。关于不同迭代器和消费者的列表，查看迭代器模块文档。

并发

> concurrency.md
>
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

并发与并行是计算机科学中相当重要的两个主题，并且在当今生产环境中也十分热门。计算机正拥有越来越多的核心，然而很多程序员还没有准备好去完全的利用它们。

Rust 的内存安全功能也适用于并发环境。甚至并发的 Rust 程序也会是内存安全的，并且没有数据竞争。Rust 的类型系统也能胜任，并且在编译时能提供你强大的方式去推论并发代码。

在我们讨论 Rust 提供的并发功能之前，理解一些问题是很重要的：Rust 非常底层以至于所有这些都是由标准库，而不是由语言提供的。这意味着如果你在某些方面不喜欢 Rust 处理并发的方式，你可以自己实现一个。mio是关于这个原则实践的一个实际的例子。

背景：`Send` 和 `Sync`

并发难以推理。在 Rust 中，我们有一个强大、静态类型系统来帮助我们推理我们的代码。Rust 自身提供了两个特性来帮助我们理解可能是并发的代码的意思。

`Send`

第一个我们要谈到的特性是Send。当一个 `T` 类型实现了 `Send` ，它向编译器器指示这个类型的所有权可以在线程间安全的转移。

强制实施一些通用的限制是很重要的。例如，我们有一个连接两个线程的通道，我们想要能够向通道发送些数据到另一个线程。因此，我们要确保这个类型实现了 `Send` 。

相反的，如果我们通过 FFI 封装了一个不是线程安全的库，我们并不想实现 `Send` ，那么编译器会帮助我们强制确保它不会离开当前线程。

`Sync`

第二个特性是 `Sync` 。当一个类型 `T` 实现了 `Sync` ，它向编译器指示这个类型在多线程并发时没有导致内存不安全的可能性。这隐含了没有内部可变性的类型天生是 `Sync` 的，这包含了基本类型（如 `u8` ）和包含他们的聚合类型。

为了在线程间共享引用，Rust 提供了一个叫做 `Arc<T>` 的 wrapper 类型。 `Arc<T>` 实现了 `Send` 和 `Sync` 当且仅当 `T` 实现了 `Send` 和 `Sync` 。例如，一个 `Arc<RefCell<U>>` 类型的对象不能在线程间传送因为RefCell并没有实现 `Sync` ，因此 `Arc<RefCell<U>>` 并不会实现 `Send` 。

这两个特性允许你使用类型系统来确保你代码在并发环境的特性。在我们演示为什么之前，我们需要先学会如何创建一个并发 Rust 程序！

线程

Rust标准库提供了一个“线程”库，它允许你并行的执行 Rust 代码。这是一个使用 `std::thread` 的基本例子：

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("Hello from a thread!");
    });
}
```

`thread::spawn()` 方法接受一个闭包，它将会在一个新线程中执行。它返回一线程的句柄，这个句柄可以用来等待子线程结束并提取它的结果：

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hello from a thread!"
    });

    println!("{}", handle.join().unwrap());
}
```

正如闭包可以从它的环境中获取变量，我们也可以把一些数据带到其他线程中：

```
use std::thread;

fn main() {
    let x = 1;
    thread::spawn(|| {
        println!("x is {}", x);
    });
}
```

然而，这会给我们一个错误：

```
5:19: 7:6 error: closure may outlive the current function, but i
t
                 borrows `x`, which is owned by the current func
tion
...
5:19: 7:6 help: to force the closure to take ownership of `x` (a
nd any other referenced variables),
          use the `move` keyword, as shown:
      thread::spawn(move || {
          println!("x is {}", x);
      });
```

这是因为默认闭包通过引用获取变量，因此闭包只会获取一个 `x` 的引用。这样有一个问题，因为线程可能会存在超过 `x` 的作用域，导致产生一个悬垂指针。

为了解决这个问题，我们使用错误中提示的 `move` 闭包。 `move` 闭包将在这里详细讲解。从基本上讲，它把变量从环境中移动到自身。这意味着 `x` 现在归闭包所有，并不能在调用 `spawn()` 后的 `main()` 中使用。

```rust
use std::thread;

fn main() {
    let x = 1;
    thread::spawn(move || {
        println!("x is {}", x);
    });
}
```

很多语言有执行多线程的能力，不过是很不安全的。有完整的书籍是关于如何避免在共享可变状态下出现错误的。在此，借助类型系统，Rust也通过在编译时避免数据竞争来帮助我们。让我们具体讨论下如何在线程间共享数据。

安全共享的可变状态（Safe Shared Mutable State）

根据 Rust的 类型系统，我们有个听起来类似谎言的概念叫做：“安全共享的可变状态”。很多程序员都同意共享可变状态是非常，非常不好的。

有人曾说道：

> 共享可变状态是一切罪恶的根源。大部分语言尝试解决这个问题的“可变”部分，而Rust则尝试解决“共享”部分。

同样所有权系统也通过防止不当的使用指针来帮助我们排除数据竞争，最糟糕的并发bug之一。

作为一个例子，这是一个在很多语言中可能会产生数据竞争的 Rust 版本程序。它不能编译：

```
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

这会给我们一个错误：

```
8:17 error: capture of moved value: `data`
        data[0] += i;
        ^~~~
```

Rust 知道这并不是安全的！如果每个线程中都有一个 `data` 的引用，并且这些线程获取了引用的所有权，我们就有了 3 个所有者！ `data` 在第一次调用 `spawn()` 时被移出了 `main` ，所以循环中接下来的调用不能使用这个变量。

所以，我们需要一些类型可以让我们拥有一个值的多个有所有权的引用。通常，我们使用 `Rc<T>` ，它是一个引用计数类型用以提供共享的所有权。它有一些运行时记录来跟踪引用它的数量，也就是“引用计数”。

调用 `Rc<T>` 的 `clone()` 方法会返回一个有所有权的引用并增加其内部引用计数。我们为每一个线程创建一个：

```
use std::thread;
use std::time::Duration;
use std::rc::Rc;

fn main() {
    let mut data = Rc::new(vec![1, 2, 3]);

    for i in 0..3 {
        // Create a new owned reference:
        let data_ref = data.clone();

        // Use it in a thread:
        thread::spawn(move || {
            data_ref[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

这并不能运行，不过它会给我们这个错误：

```
13:9: 13:22 error: the trait bound `alloc::rc::Rc<collections::v
ec::Vec<i32>> : core::marker::Send`
            is not satisfied
...
13:9: 13:22 note: `alloc::rc::Rc<collections::vec::Vec<i32>>`
            cannot be sent between threads safely
```

如错误中提到的，`Rc` 并不能在线程间安全的传递。这是因为其内部的引用计数并不是通过一个线程安全的方式维护的（非原子性操作）并可能产生数据竞争。

为了解决这个问题，我们使用 `Arc<T>`，Rust 标准的原子引用计数类型。

`Arc<T>` 的原子部分可以在多线程中安全的访问。为此编译器确保了内部计数的改变都是不可分割的操作这样就不会产生数据竞争。

本质上，`Arc<T>` 是一个可以让我们在线程间安全的共享所有权的类型。

```
use std::thread;
use std::sync::Arc;
use std::time::Duration;

fn main() {
    let mut data = Arc::new(vec![1, 2, 3]);

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            data[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

与之前类似，我们使用 `clone()` 来创建一个新的有所有权的句柄。接着这个句柄被移动到了新线程。

不过。。。仍然出错了。

```
<anon>:11:24 error: cannot borrow immutable borrowed content as mutable
<anon>:11                    data[0] += i;
                             ^~~~
```

`Arc<T>` 默认是不可变的。它允许在线程间共享数据，不过可变的共享数据是不安全的并且在涉及到多线程时会造成数据竞争！

通常当我们希望让某个不可变的东西变成可变时，我们使用 `Cell<T>` 或者 `RefCell<T>`，它们通过运行时检查或其他手段提供安全的可变性。然而，与 `Rc` 类似，它们不是线程安全的。如果我们尝试使用它们，我们将会得到一个错误说这些类型不能被 `Sync`，代码会编译失败。

看起来我们需要一些允许我们安全的在线程间改变共享值的类型，例如同一时刻只允许一个线程能够改变它内部值的类型。

为此，我们可以使用 `Mutex<T>` 类型！

下面是一个可以工作的版本：

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

注意 `i` 的值被限制（拷贝）到了闭包里并不是在线程间共享。

这里我们"锁定"了互斥锁（mutex）。一个互斥锁，正如其名，同时只允许一个线程访问一个值。当我们想要访问一个值时，我们 `lock()` 它。这会"锁定" mutex，并且其他线程不能锁定它（也就是改变它的值），直到我们处理完之后。如果一个线程尝试锁定一个已经被锁定的 mutex，它将会等待直到其他线程释放这个锁为止。

这里锁的"释放"是隐式的；当锁的结果（在这里是 `data` ）离开作用域，锁将会被自动释放。

注意Mutex的lock方法有如下签名：

```
fn lock(&self) -> LockResult<MutexGuard<T>>
```

并且因为 `MutexGuard<T>` 并没有实现 `Send` ，guard 并不能跨线程，确保了线程局部性锁的获取和释放。

让我们更仔细的检查一个线程代码：

```
# use std::sync::{Arc, Mutex};
# use std::thread;
# use std::time::Duration;
# fn main() {
#     let data = Arc::new(Mutex::new(vec![1, 2, 3]));
#     for i in 0..3 {
#         let data = data.clone();
thread::spawn(move || {
    let mut data = data.lock().unwrap();
    data[0] += i;
});
#     }
#     thread::sleep(Duration::from_millis(50));
# }
```

首先，我们调用 `lock()` ，它获取了互斥锁。因为这可能失败，它返回一个 `Result<T, E>` ，并且因为这仅仅是一个例子，我们 `unwrap()` 结果来获得一个数据的引用。现实中的代码在这里应该有更健壮的错误处理。下面我们可以随意修改它，因为我们持有锁。

最后，在线程运行的同时，我们等待在一个较短的定时器上。不过这并不理想：我们可能选择等待了一个合理的时间不过它更可能比所需的时间要久或并不足够长，这依赖程序运行时线程完成它的计算所需的时间。

一个比定时器更精确的替代是使用一个 Rust 标准库提供的用来同步各个线程的机制。让我们聊聊其中一个：通道。

通道（Channels）

下面是我们代码使用通道同步的版本，而不是等待特定时间：

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc;

fn main() {
    let data = Arc::new(Mutex::new(0));

    // `tx` is the "transmitter" or "sender".
    // `rx` is the "receiver".
    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;

            tx.send(()).unwrap();
        });
    }

    for _ in 0..10 {
        rx.recv().unwrap();
    }
}
```

我们使用 `mpsc::channel()` 方法创建了一个新的通道。我们仅仅向通道中 `send` 了一个简单的 `()` ，然后等待它们 10 个都返回。

因为这个通道只是发送了一个通用信号，我们也可以通过通道发送任何实现了 `Send` 的数据！

```rust
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = i * i;

            tx.send(answer).unwrap();
        });
    }

    for _ in 0..10 {
        println!("{}", rx.recv().unwrap());
    }
}
```

这里我们创建了 10 个线程，分别计算一个数字的平方（ `spawn()` 时的 `i` ），接着通过通道把结果 `send()` 回主线程。

恐慌（Panics）

`panic!` 会使当前执行线程崩溃。你可以使用 Rust 的线程来作为一个简单的隔离机制：

```
use std::thread;

let handle = thread::spawn(move || {
    panic!("oops!");
});

let result = handle.join();

assert!(result.is_err());
```

我们的 `Thread` 返回一个 `Result` ,它允许我们检查我们的线程是否发生了恐慌。

错误处理

> error-handling.md
>
> commit e89eee11744dee6dd13ff146817cdd4075952a0a

就像大多数编程语言，Rust 鼓励程序猿以特定的方式处理错误。一般来讲，错误处理被分割为两个大类：异常和返回值。Rust 选择了返回值。

在这一部分，我们试图提供一个全面的 Rust 如何处理错误的解决方案。不仅如此，我们也尝试一次一点的介绍错误处理，这样当你离开时会有一个对所有东西如何协调的坚实理解。

Rust 的错误处理天生是冗长而烦人的。这一部分将会探索这些坑并展示如何使用标准库来让错误处理变得准确和符合工程原理。

内容列表

这一部分灰常的长，大部分因为我们从最基础的和类型和组合入手，并尝试一点一点的解释 Rust 错误处理的动机。为此，对有其他类似类型系统经验的童鞋可能想要跳过一些内容。

基础

你可以认为错误处理是用事例分析（case analysis）来决定一个计算成功与否。如你所见，工程性的错误处理就是要减少程序猿显式的事例分析的同时保持代码的可组合性。

保持代码的可组合性是很重要的，因为没有这个要求，我们可能在遇到没想到的情况时panic。（`panic` 导致当前线程结束，而在大多数情况，导致整个程序结束。）这是一个例子：

```
// Guess a number between 1 and 10.
// If it matches the number we had in mind, return `true`. Else,
 return `false`.
fn guess(n: i32) -> bool {
    if n < 1 || n > 10 {
        panic!("Invalid number: {}", n);
    }
    n == 5
}

fn main() {
    guess(11);
}
```

如果你运行这段代码，程序会崩溃并输出类似如下信息：

```
thread 'main' panicked at 'Invalid number: 11', src/bin/panic-si
mple.rs:5
```

这是另一个稍微不那么违和的例子。一个接受一个整型作为参数，乘以二并打印的程序。

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // error 1
    let n: i32 = arg.parse().unwrap(); // error 2
    println!("{}", 2 * n);
}
```

如果你给这个程序 0 个参数（错误 1）或者 第一个参数并不是整型（错误 2），这个程序也会像第一个例子那样 panic。

你可以认为这种风格的错误处理类似于冲进瓷器店的公牛。它会冲向任何它想去的地方，不过会毁掉过程中的一切。

理解 unwrapping

在之前的例子中，我们声称程序如果遇到两个错误情况之一会直接 panic，不过，程序并不像第一个程序那样包括一个显式的 `panic` 调用。这是因为 panic 嵌入到了 `unwrap` 的调用中。

Rust 中"unwrap"是说，"给我计算的结果，并且如果有错误，panic 并停止程序。"因为他们很简单如果我们能展示 unwrap 的代码就更好了，不过在这么做之前，我们首先需要探索 `Option` 和 `Result` 类型。他们俩都定义了一个叫 `unwrap` 的方法。

`Option` 类型

`Option` 类型定义在标准库中：

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option` 类型是一个 Rust 类型系统用于表达不存在的可能性（*possibility of absence*）的方式。将不存在的可能性编码进类型系统是一个重要概念，因为它会强迫编译器处理不存在的情况。让我们看看一个尝试在一个字符串中找一个字符的例子：

```
// Searches `haystack` for the Unicode character `needle`. If on
e is found, the
// byte offset of the character is returned. Otherwise, `None` i
s returned.
fn find(haystack: &str, needle: char) -> Option<usize> {
    for (offset, c) in haystack.char_indices() {
        if c == needle {
            return Some(offset);
        }
    }
    None
}
```

注意当函数找到一个匹配的字符，它并不仅仅返回 `offset` 。相反，它返回 `Some(offset)` 。 `Some` 是一个 `Option` 类型的一个变体或一个值构造器。你可以认为它是一个 `fn<T>(value: T) -> Option<T>` 类型的函数。同理， `None` 也是一个值构造器，除了它并没有参数。你可以认为 `None` 是一个 `fn<T>() -> Option<T>` 类型的函数。

这可能看起来并没有什么，不过这是故事的一半。另一半是使用我们编写的 `find` 函数。让我们尝试用它查找文件名的扩展名。

```
# fn find(haystack: &str, needle: char) -> Option<usize> { haystack.find(needle) }
fn main() {
    let file_name = "foobar.rs";
    match find(file_name, '.') {
        None => println!("No file extension found."),
        Some(i) => println!("File extension: {}", &file_name[i+1..]),
    }
}
```

这段代码使用模式识别来对 `find` 函数的返回的 `Option<usize>` 进行 case analysis。事实上，case analysis 是唯一能获取 `Option<T>` 中存储的值的方式。这意味着你，作为一个程序猿，必须处理当 `Option<T>` 是 `None` 而不是 `Some(t)` 的情况。

不过稍等，那我们之前使用的 `unwrap` 呢？那里并没有 case analysis！相反，case analysis 被放入了 `unwrap` 方法中。如果你想的话你可以自己定义它：

```
enum Option<T> {
    None,
    Some(T),
}

impl<T> Option<T> {
    fn unwrap(self) -> T {
        match self {
            Option::Some(val) => val,
            Option::None =>
              panic!("called `Option::unwrap()` on a `None` valu
e"),
        }
    }
}
```

`unwrap` 方法抽象出了 case analysis。这正是 `unwrap` 的工程化用法。不幸的是，`panic!` 意味着 `unwrap` 并不是可组合的：它是瓷器店中的公牛。

组合 `Option<T>` 值

在之前的例子中，我们看到了如何用 `find` 发现文件名的扩展名。当然，并不是所有文件名都有一个 `.` ，所以可能文件名并没有扩展名。不存在的可能性被编码进了使用 `Option<T>` 的类型。换句话说，编译器将会强制我们描述一个扩展名不存在的可能性。在我们的例子中，我们只打印出一个说明情况的信息。

获取一个文件名的扩展名是一个很常见的操作，所以把它放进一个函数是很有道理的：

```
# fn find(haystack: &str, needle: char) -> Option<usize> { haystack.find(needle) }
// Returns the extension of the given file name, where the extension is defined
// as all characters following the first `.`.
// If `file_name` has no `.`, then `None` is returned.
fn extension_explicit(file_name: &str) -> Option<&str> {
    match find(file_name, '.') {
        None => None,
        Some(i) => Some(&file_name[i+1..]),
    }
}
```

（专业建议：不要用这段代码，相反使用标准库的extension方法。）

代码是简单的，不过重要的是注意到 `find` 的类型强迫我们考虑不存在的可能性。这是一个好事，因为这意味着编译器不会让我们不小心忘记了文件名没有扩展名的情况。另一方面，每次都像 `extension_explicit` 那样进行显式 case analysis 会变得有点无聊。

事实上，`extension_explicit` 的 case analysis 遵循一个非常常见的模式：将 `Option<T>` 中的值映射为一个函数，除非它是 `None` ，这时，返回 `None` 。

Rust 拥有参数多态（parametric polymorphism），所以定义一个组合来抽象这个模式是很容易的：

```
fn map<F, T, A>(option: Option<T>, f: F) -> Option<A> where F: FnOnce(T) -> A {
    match option {
        None => None,
        Some(value) => Some(f(value)),
    }
}
```

事实上，`map` 是标准库中的 `Option<T>` 定义的一个方法。值得一提的是，它有着一个略微不同的 signature：一个以 `self` ，`&self` 或 `&mut self` 作为第一个参数的方法。

用我们的新组合，我们可以重写我们的 `extension_explicit` 方法来去掉 case analysis：

```
# fn find(haystack: &str, needle: char) -> Option<usize> { haystack.find(needle) }
// Returns the extension of the given file name, where the extension is defined
// as all characters following the first `.`.
// If `file_name` has no `.`, then `None` is returned.
fn extension(file_name: &str) -> Option<&str> {
    find(file_name, '.').map(|i| &file_name[i+1..])
}
```

我们通常会发现的另一个模式是为一个 `Option` 为 `None` 时赋一个默认值。例如，也许你的程序假设即便一个文件没有扩展名则它的扩展名是 `rs` 。正如你可能想象到的，这里的 case analysis 并不特定用于文件扩展名 - 它可以用于任何 `Option<T>` ：

```
fn unwrap_or<T>(option: Option<T>, default: T) -> T {
    match option {
        None => default,
        Some(value) => value,
    }
}
```

与上面的 `map` 相似，标准库中的实现是一个方法而不是一个普通的函数。

这里要注意的是默认值的类型必须与可能出现在 `Option<T>` 中的值类型相同。在我们的例子中使用它是非常简单的：

```
# fn find(haystack: &str, needle: char) -> Option<usize> {
#     for (offset, c) in haystack.char_indices() {
#         if c == needle {
#             return Some(offset);
#         }
#     }
#     None
# }
#
# fn extension(file_name: &str) -> Option<&str> {
#     find(file_name, '.').map(|i| &file_name[i+1..])
# }
fn main() {
    assert_eq!(extension("foobar.csv").unwrap_or("rs"), "csv");
    assert_eq!(extension("foobar").unwrap_or("rs"), "rs");
}
```

（注意 `unwrap_or` 是标准库中的 `Option<T>` 定义的一个方法，所以这里我们使用它而不是我们上面定义的独立的函数。别忘了看看更通用的 `unwrap_or_else` 方法。）

还有另一个我们认为值得特别注意的组合：`and_then` 。它让我们更容易地组合不同的代码来应对不存在的可能。例如，这一部分的很多代码是关于找到一个给定文件的扩展名的。为此，你首先需要一个通常截取自文件路径的文件名。虽然大部分文件路径都有一个文件名，但并不是都有。例如，`.` ，`..` ，`/` 。

所以，我们面临着从一个给定的文件路径找出一个扩展名的挑战。让我们从显式 case analysis 开始：

```
# fn extension(file_name: &str) -> Option<&str> { None }
fn file_path_ext_explicit(file_path: &str) -> Option<&str> {
    match file_name(file_path) {
        None => None,
        Some(name) => match extension(name) {
            None => None,
            Some(ext) => Some(ext),
        }
    }
}

fn file_name(file_path: &str) -> Option<&str> {
  // Implementation elided.
  unimplemented!()
}
```

你可能认为我们应该用 `map` 组合来减少 case analysis，不过它的类型并不匹配。。。

```
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).map(|x| extension(x)) // This causes a
compilation error.
}
```

这里的 `map` 函数装箱了 `extension` 函数返回的 `Option<_>` 中的值，并且因为 `extension` 返回一个 `Option<&str>` ，表达式 `file_name(file_path).map(|x| extension(x))` 实际上返回一个 `Option<Option<&str>>` 。

不过因为 `file_path_ext` 仅仅返回 `Option<&str>` （而不是 `Option<Option<&str>>` ），我们会遇到一个编译错误。

被 map 函数作为输入的函数的返回值总是会被重新封装为 `Some` 。因此，我们需要一些像 `map` ，不过允许调用者直接返回一个 `Option<_>` 而不用在再套上另一个 `Option<_>` 的函数。

它的泛型实现甚至比 `map` 更简单：

```
fn and_then<F, T, A>(option: Option<T>, f: F) -> Option<A>
        where F: FnOnce(T) -> Option<A> {
    match option {
        None => None,
        Some(value) => f(value),
    }
}
```

现在我们可以不用显式 case analysis 重写我们的 `file_path_ext` 函数了：

```
# fn extension(file_name: &str) -> Option<&str> { None }
# fn file_name(file_path: &str) -> Option<&str> { None }
fn file_path_ext(file_path: &str) -> Option<&str> {
    file_name(file_path).and_then(extension)
}
```

边注：因为 `and_then` 本质上就像 `map` 不过返回一个 `Option<_>` 而不是 `Option<Option<_>>` ，它在一些其他语言中被称为 `flatmap` 。

`Option` 类型有很多其他定义在标准库中的组合。过一遍这个列表并熟悉他们的功能是一个好主意 —— 通常他们可以减少你的 case analysis。熟悉这些组合将会得到回报，因为他们很多也为 `Result` 类型定义了实现（相似的语义），而我们接下来会讲 `Result` 。

组合使用像 `Option` 这样的符合工程学的类型来减少显式 case analysis。他们也是可组合的因为他们允许调用者以他们自己的方式处理不存在的可能性。像 `unwrap` 这样的方法去掉了选择因为当 `Option<T>` 为 `None` 他们会 panic。

`Result` 类型

`Result` 类型也定义于标准库中：

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` 是 `Option` 的高级版本。相比于像 `Option` 那样表示不存在的可能性， `Result` 表示错误的可能性。通常，错误用来解释为什么一些计算会失败。严格的说这是一个更通用的 `Option` 。考虑如下类型别名，它的语义在任何地方都与真正的 `Option<T>` 相同：

```
type Option<T> = Result<T, ()>;
```

它把 `Result` 的第二个类型参数改为总是 `()` （读作“单元”或“空元组”）。 `()` 类型只有一个值： `()` （没错，类型和值两个级别的术语表示是一样的！）

`Result` 类型是一个代表一个计算的两个可能结果的方式。通常，一个结果是期望的值或者“ `Ok` ”而另一个意味着非预期的或者“ `Err` ”。

就像 `Option` ， `Result` 在标准库中也定义了一个 `unwrap` 方法。让我们定义它：

```
# enum Result<T, E> { Ok(T), Err(E) }
impl<T, E: ::std::fmt::Debug> Result<T, E> {
    fn unwrap(self) -> T {
        match self {
            Result::Ok(val) => val,
            Result::Err(err) =>
              panic!("called `Result::unwrap()` on an `Err` valu
e: {:?}", err),
        }
    }
}
```

这实际上与 `Option::unwrap` 的定义一样，除了它在 `panic!` 信息中包含了错误信息。这让调试变得简单，不过也要求我们为 `E` 类型参数（它代表我们的错误类型）添加一个 `Debug` 限制。因为绝大部分类型应该满足 `Debug` 限制，这使得它可以在实际中使用。（ `Debug` 简单的意味着这个类型有合理的方式可以打印出人类可读的描述。）

OK，让我们开始一个例子。

解析整型

Rust 标准库中让字符串转换为整型变得异常简单。事实上它太简单了，以至于你可以写出如下代码：

```
fn double_number(number_str: &str) -> i32 {
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let n: i32 = double_number("10");
    assert_eq!(n, 20);
}
```

在这里，你应该对调用 `unwrap` 持怀疑态度。例如，如果字符串并不能解析为一个数字，它会 panic：

```
thread 'main' panicked at 'called `Result::unwrap()` on an `Err`
 value: ParseIntError { kind: InvalidDigit }', /home/rustbuild/s
rc/rust-buildbot/slave/beta-dist-rustc-linux/build/src/libcore/r
esult.rs:729
```

这是很难堪的，而且如果这在你所使用的库中出现了的话，可以理解你会很烦躁。相反，我们应该尝试在我们的函数里处理错误并让调用者决定该怎么做。这意味着改变 `double_number` 的返回值类型。不过改编成什么呢？好吧，这需要我们看看标准库中 `parse` 方法的签名：

```
impl str {
    fn parse<F: FromStr>(&self) -> Result<F, F::Err>;
}
```

额嗯。所以至少我们知道了我们需要使用一个 `Result` 。当然，也可以返回一个 `Option` 。毕竟，一个字符串要么能解析成一个数字要么不能，不是吗？这当然是一个合理的方式，不过实现内部区别了为什么字符串不能解析成数字。（要么是一个空字符串，一个无效的数位，太大或太小。）因此，使用 `Result` 更有道理因为我们想要比单纯的“不存在”提供更多信息。我们想要表明为什么解析会失败。你

应该尝试再现这样的推理，当你面对一个 `Option` 和 `Result` 之间的选择时。如果你可以提供详细的错误信息，那么大概你也应该提供。（我们会在后面详细讲到。）

好的，不过我们的返回值类型该怎么写呢？上面定义的 `parse` 方法对所有不同的标准库定义的数字类型是泛型的。我们也可以（应该）让我们的函数也是泛型的，不过这回让我们享受显式定义的好处。我们只关心 `i32`，所以我们需要寻找 `FromStr` 的实现（在你的浏览器中用 `CTRL-F` 搜索“FromStr”）和与它相关的类型` `Err`。这么做我可以找出具体的错误类型。在这个例子中，它是 `std::num::ParseIntError`。最后我们可以重写函数：

```
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError>
 {
    match number_str.parse::<i32>() {
        Ok(n) => Ok(2 * n),
        Err(err) => Err(err),
    }
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

这比之前有些进步，不过现在我们写的代码有点多了！case analysis 又一次坑了我们。

组合是救星！就像 `Option` 一样，`Result` 有很多定义的组合方法。`Result` 和 `Option` 之间的常用组合有很大的交集。特别的，`map` 就是其中之一：

```rust
use std::num::ParseIntError;

fn double_number(number_str: &str) -> Result<i32, ParseIntError>
 {
    number_str.parse::<i32>().map(|n| 2 * n)
}

fn main() {
    match double_number("10") {
        Ok(n) => assert_eq!(n, 20),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

常见的组合 `Result` 都有，包括unwrap_or和and_then。另外，因为 `Result` 有第二个类型参数，所以有一些只影响错误类型的组合，例如map_err（相对于 `map` ）和or_else（相对于 `and_then` ）。

`Result` 类型别名习惯

在标准库中，你可能经常看到像 `Result<i32>` 这样的类型。不过等等，我们定义的 `Result` 有两个类型参数。我么怎么能只指定一个呢？这里的关键是定义一个 `Result` 类型别名来对一个特定类型固定其中一个类型参数。通常固定的类型是错误类型。例如，我们之前的解析整数例子可以重写成这样：

```rust
use std::num::ParseIntError;
use std::result;

type Result<T> = result::Result<T, ParseIntError>;

fn double_number(number_str: &str) -> Result<i32> {
    unimplemented!();
}
```

为什么我们应该这么做？好吧，如果我们有很多可能返回 `ParseIntError` 的函数，那么定义一个总是使用 `ParseIntError` 的别名就比每回都写一遍要方便很多。

这个习惯最突出的一点是标准库中的 `io::Result` 。通常，当你使用 `io::Result<T>` ，很明显你就是在使用 `io` 模块的类型别名而不是 `std::result` 的原始定义。（这个习惯也用于 `fmt::Result` 。）

小插曲：unwrapping 并不邪恶

如果你一路跟了过来，你可能注意到我们花了很大力气反对使用像 `unwrap` 这样会 `panic` 并终止你的程序的方法。通常来说，这是一个好的建议。

然而， `unwrap` 仍然可以被明智的使用。具体如何正当化 `unwrap` 的使用是一个灰色区域并且理性的人可能不会同意。我会简述这个问题的一些个人看法。

- 在例子和简单快速的编码中。有时你要写一个例子或小程序，这时错误处理一点也不重要。这种情形要击败 `unwrap` 的方便易用是很难的，所以它显得很合适。
- 当 **panic** 就意味着程序中有 **bug** 的时候。当你代码中的不变量应该阻止特定情况发生的时候（比如，从一个空的栈上弹出一个值），那么 panic 就是可行的。这是因为它暴露了程序的一个 bug。这可以是显式的，例如一个 `assert!` 失败，或者因为一个数组越界。

这可能并不是一个完整的列表。另外，当使用 `Option` 的时候，通常使用 `expect` 方法更好。 `expect` 做了 `unwrap` 同样的工作，除了 `expect` 会打印你给它的信息。这让 panic 的结果更容易处理，因为相比“called unwrap on a None value.”会提供一个信息。

我的建议浓缩如下：运用你良好的判断。我的文字中并没有出现“永远不要做 X”或“Y 被认为是有害的”是有原因的。所有这些都是权衡取舍，并且这是你们程序猿的工作去决定在你的用例中哪个是可以接受的。我们目标只是尽可能的帮助你进行权衡。

现在我们介绍了 Rust 的基本的错误处理，并解释了 unwrap，让我们开始更多的探索标准库。

处理多种错误类型

到目前为止，我看到了不是 `Option<T>` 就是 `Result<T, SomeError>` 的错误处理。不过当你同时使用 `Option` 和 `Result` 时会发生什么呢？或者如果你使用 `Result<T, Error1>` 和 `Result<T, Error2>` 呢？我们接下来的挑战是处理不

同错误类型的组合，这将会是贯穿本章余下部分的主要主题。

组合 `Option` 和 `Result`

到目前为止，我们讲到了为 `Option` 和 `Result` 定义的组合。我们可以用这些组合来处理不同的计算结果而不用进行显式的 case analysis。

当然，在实际的代码中，事情并不总是这么明显。有时你遇到一个 `Option` 和 `Result` 的混合类型。我们是必须求助于显式 case analysis，或者我们可以使用组合呢？

现在，让我们重温这一部分的第一个例子：

```
use std::env;

fn main() {
    let mut argv = env::args();
    let arg: String = argv.nth(1).unwrap(); // error 1
    let n: i32 = arg.parse().unwrap(); // error 2
    println!("{}", 2 * n);
}
```

基于我们新掌握的关于 `Option` ， `Result` 和他们的组合的知识，我们应该尝试重写它来适当的处理错误这样出错时程序就不会 panic 了。

这里的坑是 `argv.nth(1)` 产生一个 `Option` 而 `arg.parse()` 产生一个 `Result` 。他们不是直接可组合的。当同时遇到 `Option` 和 `Result` 的时候，解决办法通常是把 `Option` 转换为一个 `Result` 。在我们的例子中，缺少命令行参数（来自 `env::args()` ）意味着我们的用户没有正确调用我们的程序。我们可以用一个 `String` 来描述这个错误。让我们试试：

```
use std::env;

fn double_arg(mut argv: env::Args) -> Result<i32, String> {
    argv.nth(1)
        .ok_or("Please give at least one argument".to_owned())
        .and_then(|arg| arg.parse::<i32>().map_err(|err| err.to_
string()))
        .map(|n| 2 * n)
}

fn main() {
    match double_arg(env::args()) {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

这个例子中有几个新东西。第一个是使用了 `Option::ok_or` 组合。这是一个把 `Option` 转换为 `Result` 的方法。这个转换要求你指定当 `Option` 为 `None` 时的错误。就像我们见过的其他组合一样，它的定义是非常简单的：

```
fn ok_or<T, E>(option: Option<T>, err: E) -> Result<T, E> {
    match option {
        Some(val) => Ok(val),
        None => Err(err),
    }
}
```

另一个新使用的组合是 `Result::map_err` 。这就像 `Result::map` ，除了它映射一个函数到 `Result` 的 error 部分。如果 `Result` 是一个 `Ok(...)` ，那么它什么也不修改。

这里我们使用 `map_err` 是因为它对保证相同的错误类型（因为我们使用了 `and_then` ）是必要的。因为我们选择把 `Option<String>` （来自 `argv.nth(1)` ）转换为 `Result<String, String>` ，我们也必须把来自 `arg.parse()` 的 `ParseIntError` 转换为 `String` 。

组合的限制

IO 和 解析输入是非常常见的任务，这也是我个人在 Rust 经常做的。因此，我们将使用（并一直使用）IO 和多种解析工作作为例子讲解错误处理。

让我们从简单的开始。我们的任务是打开一个文件，读取所有的内容并把他们转换为一个数字。接着我们把它乘以 `2` 并打印结果。

虽然我们劝告过你不要用 `unwrap`，不过开始写代码的时候 `unwrap` 也是有用的。它允许你关注你的问题而不是错误处理，并且暴露出需要错误处理的点。让我们开始试试手感，再接着用更好的错误处理重构。

```rust
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> i32 {
    let mut file = File::open(file_path).unwrap(); // error 1
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap(); // error 2
    let n: i32 = contents.trim().parse().unwrap(); // error 3
    2 * n
}

fn main() {
    let doubled = file_double("foobar");
    println!("{}", doubled);
}
```

（附注：`AsRef<Path>` 被使用是因为它与 `std::fs::File::open` 有着相同的 bound。这让我们可以用任何类型的字符串作为一个文件路径。）

这里可能出现三个不同错误：

1. 打开文件出错。
2. 从文件读数据出错。
3. 将数据解析为数字出错。

头两个错误被描述为 `std::io::Error` 类型。我们知道这些因为返回类型是 `std::fs::File::open` 和 `std::io::Read::read_to_string` 。（注意他们都使用了之前描述的 `Result` 类型别名习惯。如果你点击 `Result` 类型，你将会看到这个类型别名，以及底层的 `io::Error` 类型。）第三个问题被描述为 `std::num::ParseIntError` 。特别的 `io::Error` 被广泛的用于标准库中。你会一次又一次的看到它。

让我们着手重构 `file_double` 函数。为了让这个函数可以与程序的其他组件组合，它必须不能在上述错误情况下 panic。事实上，这意味着它在任何操作失败时应该返回一个错误。我们的问题是 `file_double` 的返回类型是 `i32` ，它并没有给我们一个有效的报告错误的途径。因此，我们必须以把返回类型 `i32` 改成别的什么的开始。

我们需要决定的第一件事：我们应该用 `Option` 还是 `Result` ？当然我们可以简单的选择 `Option` 。如果出现任何错误了，我们可以简单的返回 `None` 。它可以工作并且比 panic 好多了，不过我们可以做的更好。我们应该在错误发生时传递一些细节。因为我们想要表达错误的可能性，我们应该使用 `Result<i32, E>` 。不过 `E` 应该是什么呢？因为可能出现两种不同的错误，我们需要把他们转换为一种通用类型。其中之一就是 `String` 。让我们看看这如何影响我们的代码：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Stri
ng> {
    File::open(file_path)
         .map_err(|err| err.to_string())
         .and_then(|mut file| {
              let mut contents = String::new();
              file.read_to_string(&mut contents)
                  .map_err(|err| err.to_string())
                  .map(|_| contents)
         })
         .and_then(|contents| {
              contents.trim().parse::<i32>()
                      .map_err(|err| err.to_string())
         })
         .map(|n| 2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

这些代码看起来有点难以理解。在能轻松编写这样的代码前可能需要更多一些实践。我们写代码的方式遵循跟着类型走（*following the types*）。一旦我们把 `file_double` 的返回类型改为 `Result<i32, String>` ，我们就不得不开始寻找正确的组合。在这个例子中，我们只用到了三个不同的组合： `and_then` ， `map` 和 `map_err` 。

`and_then` 被用来连接多个计算，其中每一个都有可能返回一个错误。在打开文件后，还有另外两个可能失败的计算：从文件读取和把内容解析成数字。相应地，有两个 `and_then` 的调用。

`map` 用来把一个函数用于 `Result` 的 `Ok(...)` 值。例如，最后一个 `map` 调用把 `Ok(...)` 值（它是一个 `i32` ）乘以 `2` 。如果在这之前出现了错误，这里的操作会被省略，因为 `map` 是这么定义的。

`map_err` 是让一切可以工作的关键。 `map_err` 类似 `map` ，除了它把一个函数用于 `Result` 的 `Err(...)` 值。在这个例子中，我们想要把所有的错误转换为一个类型： `String` 。因为 `io::Error` 和 `num::ParseIntError` 都实现了 `ToString` ，我们可以调用 `to_string()` 去转换他们。

说了这么多，代码仍然不好懂。掌握组合的应用是很重要的，不过他们也有限制。让我们尝试一个不同的方式：提早返回。

提早返回

我想利用前一章节的代码并用提早返回重写它。提早返回让你提前退出函数。我们不能在另一个闭包中从 `file_double` 提前返回，所以我们需要退回到显式 case analysis。

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Stri
ng> {
    let mut file = match File::open(file_path) {
        Ok(file) => file,
        Err(err) => return Err(err.to_string()),
    };
    let mut contents = String::new();
    if let Err(err) = file.read_to_string(&mut contents) {
        return Err(err.to_string());
    }
    let n: i32 = match contents.trim().parse() {
        Ok(n) => n,
        Err(err) => return Err(err.to_string()),
    };
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

理性的同学可能不同意这个比使用组合的代码更好，不过如果你并不熟悉组合方式，这些代码阅读起来更简单。它通过 `match` 和 `if let` 进行了显式 case analysis。如果错误出现了，它简单的停止执行并返回错误（通过转换为一个字符串）。

这难道不是倒退吗？之前，我们说过工程性的错误处理的关键是减少显式 case analysis，不过这里我们又退回到了显式 case analysis。这表明，有多种方式可以减少显式 case analysis。组合并不是唯一的方法。

`try!` 宏

Rust 中错误处理的基石是 `try!` 宏。`try!` 宏像组合一样抽象了 case analysis，不过不像组合，它也抽象了控制流。也就是说，它可以抽象我们之前看到的提早返回的模式。

这是一个简单化的 `try!` 宏定义：

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

（实际的定义有一点复杂。我们会在后面详述。）

使用 `try!` 宏让我们的最后的例子异常的简单。因为它为我们做了 case analysis 和提早返回，我们的代码更紧凑也更易于理解：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {}", err),
    }
}
```

根据我们 `try!` 宏的定义 `map_err` 调用仍是必要的。这是因为错误类型仍然需要被转换为 `String`。好消息是我们马上就会学到如何移除这些 `map_err` 调用！坏消息是在我们可以移除 `map_err` 调用之前我们需要更深入的学习一些标准库中的重要的 trait。

定义你自己的错误类型

在我们深入学习一些标准库错误 trait 之前，我想要通过移除之前例子中的作为错误类型的 `String` 来结束本章节。

之前我们的例子中使用 `String` 是为了方便，因为把错误转换为字符串是简单的，甚至把我们自己的类型转换为字符串也是如此。然而，把 `String` 作为你的错误有一些缺点。

第一个缺点是错误信息会倾向于另你的代码变得凌乱。把错误信息定义在别处是可能的，不顾除非你非常的（qiang）自（po）律（zheng），你很容易就会把错误信息嵌入到你的代码中。事实上，我们上一个例子就是这么做的。

第二个也是更重要的缺点是 `String` 是不完整的。也就是说，如果所有错误都转换成了字符串，那么传递给调用者的错误就变得完全不透明了。调用者对于一个 `String` 类型的错误所能作的唯一可行的事就是把它展示给用户。当然，通过观察字符串来确定错误的类型是不健壮的。（对于一个库来说这个缺点公认要比在例如程序中来的更重要。）

例如， `io::Error` 类型内嵌了一个 `io::ErrorKind` ，它是一个表示在 IO 操作期间错误信息的结构化类型。这很重要因为你可能根据错误做出不同的反应。（例如， `BrokenPipe` 错误可能意味着可以温和的退出程序，而 `NotFound` 则意味着应该返回一个错误码并向用户展示错误。）通过 `io::ErrorKind` ，调用者可以用 case analysis 检查错误的类型，这完全优于尝试从 `String` 中梳理错误的细节。

除了在我们之前从文件读取一个数字的例子中使用 `String` 作为错误类型外，我们可以定义我们自己的错误类型来用结构化数据代表错误。我们尽量不丢掉底层错误的信息以防调用者想要检视细节。

表示多种可能性的理想方法是用 `enum` 来定义我们的集合类型。在我们的例子里，错误要么是 `io::Error` 要么是 `num::ParseIntError` ，所以自然的定义如下：

```
use std::io;
use std::num;

// We derive `Debug` because all types should probably derive `D
ebug`.
// This gives us a reasonable human readable description of `Cli
Error` values.
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

修改我们代码非常简单。与其把错误转为字符串，我们简单的用相应的值构造器把错误转换为我们的 `CliError` 类型：

```
# #[derive(Debug)]
# enum CliError { Io(::std::io::Error), Parse(::std::num::ParseIntError) }
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::Parse));
    Ok(2 * n)
}

fn main() {
    match file_double("foobar") {
        Ok(n) => println!("{}", n),
        Err(err) => println!("Error: {:?}", err),
    }
}
```

这里唯一的修改是从 `map_err(|e| e.to_string())` （它把错误转换为字符串）变为 `map_err(CliError::Io)` 或者 `map_err(CliError::Parse)` 。调用者要决定报告给用户的细节的级别。实际上，使用 `String` 作为错误类型剥夺了调用者的选择，而用一个像 `CliError` 这样的 `enum` 类型除了一个描述错误的结构化数据之外还给了调用者所有的便利。

经验之谈是定义你自己的错误类型，不过必要时 `String` 也能行，特别是你在写一个程序时。如果你在编写一个库，强烈建议你定义自己的错误类型这样你就不会不必要的剥夺了调用者选择。

用于错误处理的标准库 trait

标准库定义了两个完整 trait 用于错误处理：`http://doc.rust-lang.org/std/error/trait.Error.html` 和 `std::convert::From`。`Error` 被专门设计为描述通用错误，`From` trait 更多的用于在两个不同类型值之间转换。

`Error` trait

`Error` trait 定义于标准库中：

```
use std::fmt::{Debug, Display};

trait Error: Debug + Display {
  /// A short description of the error.
  fn description(&self) -> &str;

  /// The lower level cause of this error, if any.
  fn cause(&self) -> Option<&Error> { None }
}
```

这个 trait 非常泛用因为它被设计为为所有类型实现来代表错误。它被证明对编写可组合的代码非常有帮助，正如我们后面将要看到的。这个 trait 允许你至少做如下事情：

- 获取一个错误的 `Debug` 表示。
- 获取一个错误的面向用户的 `Display` 表示
- 获取一个错误的简短描述（通过 `description` 方法）
- 查看错误的调用链，如果存在的话（通过 `cause` 方法）

头两个是因为 `Error` 要求实现 `Debug` 和 `Display`。后两个来自于定义于 `Error` 的方法。`Error` 的力量来自于所有错误类型都实现了 `Error` 的事实，这意味着错误可以被量化一个trait 对象。表现为 `Box<Error>` 或 `&Error`。事实上, `cause` 返回一个 `&Error`，它自身就是一个 trait 对象。我们将在后面再次讨论 `Error` 作为 trait 对象的功能。

目前，展示一个实现了 `Error` trait 的例子是足够的。让我们使用上一部分我们定义的错误类型：

```
use std::io;
use std::num;

// We derive `Debug` because all types should probably derive `D
ebug`.
// This gives us a reasonable human readable description of `Cli
Error` values.
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}
```

这个特定的错误类型表示出现两种错误类型的可能性：一个进行 I/O 操作的错误或者一个把字符串转换为数字的错误。这个类型可以表示任何你想要添加的错误类型，通过向 `enum` 定义添加变量。

实现 `Error` 是非常直观的。这会有很多的显式 case analysis。

```
use std::error;
use std::fmt;

impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            // Both underlying errors already impl `Display`, so
 we defer to
            // their implementations.
            CliError::Io(ref err) => write!(f, "IO error: {}", e
rr),
            CliError::Parse(ref err) => write!(f, "Parse error:
{}", err),
        }
    }
}

impl error::Error for CliError {
    fn description(&self) -> &str {
        // Both underlying errors already impl `Error`, so we de
```

```
fer to their
        // implementations.
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Parse(ref err) => err.description(),
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // N.B. Both of these implicitly cast `err` from the
ir concrete
            // types (either `&io::Error` or `&num::ParseIntErro
r`)
            // to a trait object `&Error`. This works because bo
th error types
            // implement `Error`.
            CliError::Io(ref err) => Some(err),
            CliError::Parse(ref err) => Some(err),
        }
    }
}
```

我们注意到这是一个非常典型的 `Error` 的实现：为你不同的错误类型做匹配并满足 `description` 和 `cause` 定义的限制。

`From` trait

`std::convert::From` trait 定义于标准库中：

```
trait From<T> {
    fn from(T) -> Self;
}
```

非常简单吧？ `From` 很有用因为它给了我们一个通用的方式来处理从一个特定类型 `T` 到其他类型的转换（在这个例子中，“其他类型”是实现的主体，或者 `Self` ）。 `From` 的核心是标准库提供的一系列实现。

这里是几个展示 `From` 如何工作的小例子：

```
let string: String = From::from("foo");
let bytes: Vec<u8> = From::from("foo");
let cow: ::std::borrow::Cow<str> = From::from("foo");
```

好的，这么说 `From` 用来处理字符串转换，那么错误怎么办？原来有一个关键实现：

```
impl<'a, E: Error + 'a> From<E> for Box<Error + 'a>
```

这个实现说任何实现了 `Error` 的类型，我们可以把它转换一个 trait 对象 `Box<Error>` 。这可能看起来并不怎么令人吃惊，不过它在泛型环境中很有用。

记得我们之前处理的两个错误吗？ `io::Error` 和 `num::ParseIntError` 。因为他们都实现了 `Error` ，他们也能用于 `From` ：

```
use std::error::Error;
use std::fs;
use std::io;
use std::num;

// We have to jump through some hoops to actually get error valu
es:
let io_err: io::Error = io::Error::last_os_error();
let parse_err: num::ParseIntError = "not a number".parse::<i32>(
).unwrap_err();

// OK, here are the conversions:
let err1: Box<Error> = From::from(io_err);
let err2: Box<Error> = From::from(parse_err);
```

这里有一个非常重要的模式。 `err1` 和 `err2` 有着相同的类型。这是因为他们实际上是定量类型，或者 trait 对象。尤其是，对编译器来说他们的底层类型被抹掉了，所以编译器认为 `err1` 和 `err2` 是完全一样的。另外，我们用完全一样的函

数调用构建 `err1` 和 `err2` ： `From::from` 。这是因为 `From::from` 的参数和返回值都可以重载。

这个模式很重要，因为它解决了一个我们之前遇到过的问题：它给了我们一个可靠的用相同的函数把错误转换为相同类型的方法。

是时候重新看看我们的老朋友： `try!` 宏了。

真正的 `try!` 宏

之前我们展示了 `try!` 的定义：

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(err),
    });
}
```

这并不是它真正的定义。它的实际定义位于标准库中：

```
macro_rules! try {
    ($e:expr) => (match $e {
        Ok(val) => val,
        Err(err) => return Err(::std::convert::From::from(err)),
    });
}
```

这是一个很小但很有效的修改：错误值被通过 `From::from` 传递。这让 `try!` 宏变得更强大了一点，因为它免费提供给你自动类型转换。

有了更强大的 `try!` 宏的支持，让我们再看一眼我们之前写的读一个文件并把内容转换为数字的代码：

```
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, String> {
    let mut file = try!(File::open(file_path).map_err(|e| e.to_string()));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(|e| e.to_string()));
    let n = try!(contents.trim().parse::<i32>().map_err(|e| e.to_string()));
    Ok(2 * n)
}
```

之前，我们承诺我们可以去掉 `map_err` 调用。实际上，所有我们需要做的就是选一个可以用于 `From` 的类型。一如我们在上一个部分看到的， `From` 有一个可以转换任意错误类型为 `Box<Error>` 的实现：

```
use std::error::Error;
use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, Box<Error>> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n = try!(contents.trim().parse::<i32>());
    Ok(2 * n)
}
```

我们已经非常接近理想的错误处理了。我们的代码处理错误只造成了很小的成本，因为 `try!` 宏同时封装了三个东西：

1. case analysis。

2. 控制流。
3. 错误类型转换。

当结合所有这些东西，我们的代码不再受组合、`unwrap` 调用或 case analysis 的困扰了。

这里还剩一点东西：`Box<Error>` 是不透明的。如果我们返回一个 `Box<Error>` 给调用者，调用者并不能（轻易地）观察底层错误类型。当然这种情形比 `String` 要好，因为调用者可以调用像 `description` 和 `cause` 这样的方法，不过这是有限制的：`Box<Error>` 是不透明的。（附注：这并不是完全正确，因为 Rust 并没有运行时反射，这在某些场景是有用的不过超出了本部分的范畴。）

是时候重写我们的 `CliError` 类型并将一切连起来了。

组合自定义错误类型

在这最后一部分，我们看看真正的 `try!` 宏以及如何通过调用 `From::from` 自动转换错误类型。具体的，我们把错误转换为 `Box<Error>`，这是可以的，不过这个类型对调用者是不透明的。

为了修改这个问题，我们使用我们已经熟知的补救方法：一个自定义错误类型。再一次，这是读取文件内容并将其转换为数字的代码：

```
use std::fs::File;
use std::io::{self, Read};
use std::num;
use std::path::Path;

// We derive `Debug` because all types should probably derive `D
ebug`.
// This gives us a reasonable human readable description of `Cli
Error` values.
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Parse(num::ParseIntError),
}

fn file_double_verbose<P: AsRef<Path>>(file_path: P) -> Result<i
32, CliError> {
    let mut file = try!(File::open(file_path).map_err(CliError::
Io));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents).map_err(CliError::Io
));
    let n: i32 = try!(contents.trim().parse().map_err(CliError::
Parse));
    Ok(2 * n)
}
```

注意我们仍然有 `map_err` 的调用。为神马？好吧，回忆 `try!` 和 `From` 的定义。问题是这里并没有 `impl` 的实现允许我们将一些像 `io::Error` 和 `num::ParseIntError` 这样的错误类型转换为我们的自定义类型 `CliError` 。当然，这个问题很好修改！ `CliError` 都是我们定义的，我们可以为其实现 `From` 。

```
# #[derive(Debug)]
# enum CliError { Io(io::Error), Parse(num::ParseIntError) }
use std::io;
use std::num;

impl From<io::Error> for CliError {
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}

impl From<num::ParseIntError> for CliError {
    fn from(err: num::ParseIntError) -> CliError {
        CliError::Parse(err)
    }
}
```

所有这些实现都是告诉 `From` 如何从其他类型创建一个 `CliError` 。在我们的例子中，构造函数就像调用相应的值构造器那样简单。确实，这通常很简单。

最后我们可以重写 `file_double` ：

```
# use std::io;
# use std::num;
# enum CliError { Io(::std::io::Error), Parse(::std::num::ParseI
ntError) }
# impl From<io::Error> for CliError {
#     fn from(err: io::Error) -> CliError { CliError::Io(err) }
# }
# impl From<num::ParseIntError> for CliError {
#     fn from(err: num::ParseIntError) -> CliError { CliError::P
arse(err) }
# }

use std::fs::File;
use std::io::Read;
use std::path::Path;

fn file_double<P: AsRef<Path>>(file_path: P) -> Result<i32, CliE
rror> {
    let mut file = try!(File::open(file_path));
    let mut contents = String::new();
    try!(file.read_to_string(&mut contents));
    let n: i32 = try!(contents.trim().parse());
    Ok(2 * n)
}
```

我们做的唯一一件事就是去掉了 `map_err` 调用。他们不再必要因为 `try!` 宏对错误类型调用了 `From::from` 。这一切可以工作因为我们对所有可能出现的错误类型提供了 `From` 实现。

如果我们修改我们的 `file_double` 函数来进行一些其他操作，例如，把字符串转换为浮点数，那么我们需要给我们的错误类型增加一个新变量：

```
use std::io;
use std::num;

enum CliError {
    Io(io::Error),
    ParseInt(num::ParseIntError),
    ParseFloat(num::ParseFloatError),
}
```

并增加一个新的 `From` 实现：

```
# enum CliError {
#     Io(::std::io::Error),
#     ParseInt(num::ParseIntError),
#     ParseFloat(num::ParseFloatError),
# }

use std::num;

impl From<num::ParseFloatError> for CliError {
    fn from(err: num::ParseFloatError) -> CliError {
        CliError::ParseFloat(err)
    }
}
```

一切搞定！

给库编写者的建议

如果你的库需要报告一些自定义错误，那么你可能应该定义你自己的错误类型。由你决定是否暴露它的表示（例如 `ErrorKind` ）或者把它隐藏起来（例如 `ParseIntError` ）。不过你怎么做，相比 `String` 表示多少提供一些关于错误的信息通常是好的实践。不过说实话，这根据使用情况大有不同。

最少，你可能应该实现 `Error` trait。这会给你的库的用户以处理错误的最小灵活性。实现 `Error` trait 也意味着可以确保用户能够获得一个错误的字符串表示（因为它实现了 `fmt::Debug` 和 `fmt::Display` ）。

不仅如此，为你的错误类型提供 `From` 实现也是很有用的。这允许你（库作者）和你的用户组合更详细的错误。例如，`csv::Error` 提供了 `io::Error` 和 `byteorder::Error` 。

最后，根据你的风格，你也许想要定义一个 `Result` 类型别名，尤其是如果你的库定义了一个单一的错误类型。这被用在了标准库的 `io::Result` 和 `fmt::Result` 中。

案例学习：一个读取人口数据的程序

这一部分很长，并且根据你的背景，它可能显得更加复杂。虽然有很多示例代码以及散文一样的解释，但大部分都被设计为教科书式的。那么，我们要开始点新东西了：一个案例学习。

为此，为此我们将要建立一个可以让你查询真实世界人口数据的命令行程序。目标是简单的：你给出一个地点接着它会告诉你人口。虽然这很简单，但仍有很多地方我们可能犯错。

我们将使用的数据来自Data Science Toolkit。我为这个练习准备了一些数据。你要么可以获取世界人口数据（41 MB gzip 压缩，145 MB 未压缩）或者只使用US 人口数据（2.2 MB gzip 压缩，7.2 MB 未压缩）。

直到目前为止，我们的代码一直限制在 Rust 标准库之内。但是对于一个像这样的真实的任务，我们至少想要一些解析 CSV 数据，解析程序参数以及将其自动转换为 Rust 类型的东西。为此，我们将使用 `csv` ，以及 `rustc-serialize` crate。

初始化

我们不打算花很多时间在使用 Cargo 创建一个项目上，因为这在 Cargo 部分和 Cargo 文档中已被讲解。

为了从头开始，运行 `cargo new --bin city-pop` 并确保你的 `Cargo.toml` 看起来像这样：

```
[package]
name = "city-pop"
version = "0.1.0"
authors = ["Andrew Gallant <jamslam@gmail.com>"]

[[bin]]
name = "city-pop"

[dependencies]
csv = "0.*"
rustc-serialize = "0.*"
getopts = "0.*"
```

你应该已经可以运行了：

```
cargo build --release
./target/release/city-pop
# Outputs: Hello, world!
```

参数解析

让我们搞定参数解析，我们不会涉及太多关于 Getopts 的细节，不过有一些不错的文档。简单的说就是 Getopts 生成了一个参数解析器并通过要给选项的 vector（事实是一个隐藏于一个结构体和一堆方法之下的 vector）生成了一个帮助信息。一旦解析结束，解析器返回一个记录了匹配到定义项内容的结构体，和剩下“自由”的参数。从这里我们可以互获取 flag，实例，任何程序传递给我们的，以及他们都有什么参数。这是我们的程序，它有合适的 `extern crate` 语句以及 Getopts 的基本参数操作：

```
extern crate getopts;
extern crate rustc_serialize;

use getopts::Options;
use std::env;

fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <dat
a-path> <city>", program)));
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = &args[0];

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");

    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
    };
    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }
    let data_path = &matches.free[0];
    let city: &str = &matches.free[1];

    // Do stuff with information.
}
```

首先，我们获取一个传递给我们程序的 vector。接着我们我们储存第一个参数，因为我们知道那是程序名。当一切搞定，我们设置我们的参数 flag，在这里是一个简单的提示信息 flag。当我们设置了参数 flag 之后，我们使用 `Options.parse` 解析参数列表（从 1 开始，因为 0 是程序名）。如果这成功了，我们被解析的对象赋值给 `matches` ，如果失败了，我们 panic。接着，我们检查用户是否传递了帮助

flag，如果是就打印使用帮助信息。帮助信息选项是 Getopts 构建的，所以为了打印用法信息所有我们需要做的就是告诉它我们想要打印什么名字和模板。如果用户并没有传递帮助 flag，我们把相应的参数赋值给合适的变量。

编写逻辑

每个人写代码的方式各有不同，不过一般错误处理都是我们最后会思考的事情。这对程序整体的设计并不好，不过它对快速原型有帮助。因为 Rust 强制我们进行显示的错误处理（通过让我们调用 `unwrap` ），这样很容易看出我们的程序的那一部分可以造成错误。

在这个案例学习中，逻辑真的很简单。所有我们要做的就是解析给我们的 CSV 数据并打印出匹配的行的一个字段。让我们开始吧。（确保在你的文件开头加上 `extern crate csv;` 。）

```
use std::fs::File;

// This struct represents the data in each row of the CSV file.
// Type based decoding absolves us of a lot of the nitty gritty
error
// handling, like parsing strings as integers or floats.
#[derive(Debug, RustcDecodable)]
struct Row {
    country: String,
    city: String,
    accent_city: String,
    region: String,

    // Not every row has data for the population, latitude or lo
ngitude!
    // So we express them as `Option` types, which admits the po
ssibility of
    // absence. The CSV parser will fill in the correct value fo
r us.
    population: Option<u64>,
    latitude: Option<f64>,
    longitude: Option<f64>,
}
```

```
fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <dat
a-path> <city>", program)));
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = &args[0];

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");

    let matches = match opts.parse(&args[1..]) {
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
    };

    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }

    let data_path = &matches.free[0];
    let city: &str = &matches.free[1];

    let file = File::open(data_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);

    for row in rdr.decode::<Row>() {
        let row = row.unwrap();

        if row.city == city {
            println!("{}, {}: {:?}",
                row.city, row.country,
                row.population.expect("population count"));
        }
    }
}
```

让我们概括下错误。我们可以从明显的开始：三个 `unwrap` 被调用的地方：

1. `File::open` 可能返回 `io::Error` 。
2. `csv::Reader::decode` 一次解码一行，而且解码一个记录（查看 `Iterator` 实现的关联类型 `Item` ）可能产生一个 `csv::Error` 。
3. 如果 `row.population` 是 `None` ，那么调用 `expect` 会 panic。

还有其他的吗？如果我们无法找到一个匹配的城市呢？想 `grep` 这样的工具会返回一个错误码，所以可能我们也应该这么做。所以我们有特定于我们的问题，IO 错误和 CSV 解析错误的逻辑错误。我们将探索两个不同方式来处理这个问题。

我像从 `Box<Error>` 开始。接着，我们看看如何定义有用的自定义错误类型。

使用 `Box<Error>` 处理错误

`Box<Error>` 的好处是它刚刚够用。你并不需要定义你自己的错误类型而且也不需要任何 `From` 实现。缺点是因为 `Box<Error>` 是一个 trait 对象，这意味着编译器无法再推导出底层类型。

之前我们开始了把我们函数类型从 `T` 变成 `Result<T, OurErrorType>` 的重构。在这个例子中， `OurErrorType` 就是 `Box<Error>` 。不过 `T` 是什么？或者我们可以给 `main` 添加一个返回类型吗？

第二个问题的答案是不行，我们不能这么做。这意味着我们需要写一个新函数。不过 `T` 是什么？最简单的办法是返回一个作为 `Vec<Row>` 的匹配上的 `Row` 的值。（更好的代码会返回一个迭代器，不过这是一个留给读者的练习。）

让我们重构函数，不过保持对 `unwrap` 的调用。注意我们选择处理一个不存在的人口数行的方式是单纯的忽略它。

```
use std::path::Path;

struct Row {
    // This struct remains unchanged.
}

struct PopulationCount {
    city: String,
    country: String,
```

```
    // This is no longer an `Option` because values of this type
 are only
    // constructed if they have a population count.
    count: u64,
}

fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <dat
a-path> <city>", program)));
}

fn search<P: AsRef<Path>>(file_path: P, city: &str) -> Vec<Popul
ationCount> {
    let mut found = vec![];
    let file = File::open(file_path).unwrap();
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = row.unwrap();
        match row.population {
            None => { } // Skip it.
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    found
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let program = &args[0];

    let mut opts = Options::new();
    opts.optflag("h", "help", "Show this usage message.");

    let matches = match opts.parse(&args[1..]) {
```

```
        Ok(m)  => { m }
        Err(e) => { panic!(e.to_string()) }
    };
    if matches.opt_present("h") {
        print_usage(&program, opts);
        return;
    }

    let data_path = &matches.free[0];
    let city: &str = &matches.free[1];
    for pop in search(data_path, city) {
        println!("{}, {}: {:?}", pop.city, pop.country, pop.coun
t);
    }
}
```

虽然我们去掉了一个 `expect` 调用（它是一个比 `unwrap` 要好的变体），我们仍要处理任何不存在的搜索结果。

为了把这转化为合适的错误处理，我们需要做如下事情：

1. 把 `search` 的返回值类型改为 `Result<Vec<PopulationCount>, Box<Error>>`.
2. 使用 `try!` 宏这样会返回错误给调用者而不是使程序 panic。
3. 处理 `mian` 中的错误。

让我们试试：

```
use std::error::Error;

// The rest of the code before this is unchanged.

fn search<P: AsRef<Path>>
         (file_path: P, city: &str)
         -> Result<Vec<PopulationCount>, Box<Error>> {
    let mut found = vec![];
    let file = try!(File::open(file_path));
    let mut rdr = csv::Reader::from_reader(file);
    for row in rdr.decode::<Row>() {
        let row = try!(row);
        match row.population {
            None => { } // Skip it.
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    if found.is_empty() {
        Err(From::from("No matching cities with a population wer
e found."))
    } else {
        Ok(found)
    }
}
```

现在我们用 `try!(x)` 代替了 `x.unwrap()` 。因为我们的函数返回一个 `Result<T, E>` ， `try!` 宏在出现错误时会提早返回。

在 `search` 的结尾我们也用了相应的 `From` 实现把一个字符串转换为一个错误类型：

```
// We are making use of this impl in the code above, since we ca
ll `From::from`
// on a `&'static str`.
impl<'a> From<&'a str> for Box<Error>

// But this is also useful when you need to allocate a new strin
g for an
// error message, usually with `format!`.
impl From<String> for Box<Error>
```

因为 `search` 现在返回 `Result<T, E>` ， `main` 应该在调用 `search` 时使用 case analysis：

```
...
    match search(data_path, city) {
        Ok(pops) => {
            for pop in pops {
                println!("{}, {}: {:?}", pop.city, pop.country,
pop.count);
            }
        }
        Err(err) => println!("{}", err)
    }
...
```

现在你看到了我们如何正确的处理 `Box<Error>` ，让我们尝试一种使用我们自定义错误类型的不同方式。不过首先，让我们先放下错误处理并快速的添加从 `stdin` 读取的功能。

从标准输入读取

在我们的程序中，我们接受一个单文件输入并进行一次数据解析。这意味着我们可能需要能够接受标准输入。不过你可能也喜欢现在的格式——所以让我们同时拥有两者吧！

添加标准输入支持是非常简单的。我们只必需做三件事：

1. 修改程序参数，这样一个单独的参数——城市——可以被接受，同时人口数据

从标准输入读取。
2. 修改程序，这样一个 `-f` 选项可以接受文件，如果它没有从标准输入传递。
3. 修改 `search` 函数接受一个可选的文件路径。当为 `None` 时，它应该知道从标准输入读取。

首先，这是新的使用方法函数：

```
fn print_usage(program: &str, opts: Options) {
    println!("{}", opts.usage(&format!("Usage: {} [options] <city>", program)));
}
```

当然我们需要加入参数处理的代码：

```
...
    let mut opts = Options::new();
    opts.optopt("f", "file", "Choose an input file, instead of using STDIN.", "NAME");
    opts.optflag("h", "help", "Show this usage message.");
    ...
    let data_path = matches.opt_str("f");

    let city = if !matches.free.is_empty() {
        &matches.free[0]
    } else {
        print_usage(&program, opts);
        return;
    };

    match search(&data_path, city) {
        Ok(pops) => {
            for pop in pops {
                println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
            }
        }
        Err(err) => println!("{}", err)
    }
...
```

我们已经通过显示使用方法信息来提高了用户体验，而不是因为 `city` 这个额外的参数不存在导致索引越界而 panic，

修改 `search` 需要一点技巧。`csv` crate 可以用任何实现了 `io::Read` 的类型构建一个解析器。不过我们如何对这两个类型（注：因该是 `Option` 的两个值）使用相同的代码呢？事实上有多种方法可以做到。其中之一是重写 `search` 为接受一个满足 `io::Read` 的 `R` 类型参数的泛型。另一个办法是使用 trait 对象：

```
use std::io;

// The rest of the code before this is unchanged.

fn search<P: AsRef<Path>>
         (file_path: &Option<P>, city: &str)
         -> Result<Vec<PopulationCount>, Box<Error>> {
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(try!(File::open(file_pat
h))),
    };
    let mut rdr = csv::Reader::from_reader(input);
    // The rest remains unchanged!
}
```

用自定义类型处理错误

之前，我们学习了如何用自定义错误类型组合错误。我们定义了一个 `enum` 的错误类型并实现了 `Error` 和 `From` 。

因为我们有三个不同的错误（IO，CSV 解析和未找到），让我们定义一个三个变体的 `enum` ：

```
#[derive(Debug)]
enum CliError {
    Io(io::Error),
    Csv(csv::Error),
    NotFound,
}
```

现在让我们实现 `Display` 和 `Error` ：

```
use std::fmt;

impl fmt::Display for CliError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CliError::Io(ref err) => err.fmt(f),
            CliError::Csv(ref err) => err.fmt(f),
            CliError::NotFound => write!(f, "No matching cities
with a \
                                             population were fou
nd."),
        }
    }
}

impl Error for CliError {
    fn description(&self) -> &str {
        match *self {
            CliError::Io(ref err) => err.description(),
            CliError::Csv(ref err) => err.description(),
            CliError::NotFound => "not found",
        }
    }

    fn cause(&self) -> Option<&Error> {
        match *self {
            CliError::Io(ref err) => Some(err),
            CliError::Csv(ref err) => Some(err),
            // Our custom error doesn't have an underlying cause,

            // but we could modify it so that it does.
            CliError::NotFound => None,
        }
    }
}
```

在我们可以在 `search` 函数中使用 `CliError` 之前，我们需要提供一系列的 `From` 实现。我们如何知晓该提供那个实现呢？好吧，我们得把 `io::Error` 和 `csv::Error` 都转换为 `CliError` 。他们都只是外部错误，所以目前我们只需要两个 `From` 实现：

```
impl From<io::Error> for CliError {
    fn from(err: io::Error) -> CliError {
        CliError::Io(err)
    }
}

impl From<csv::Error> for CliError {
    fn from(err: csv::Error) -> CliError {
        CliError::Csv(err)
    }
}
```

因为 `try!` 的定义 `From` 的实现是很重要的。尤其是在这个例子中，如果出现错误，错误的 `From::from` 被调用，将被转换为我们的错误类型 `CliError` 。

当实现了 `From` ，我们只需要对 `search` 函数进行两个小的修改：返回值类型和“未找到”错误。这是全部的代码：

```
fn search<P: AsRef<Path>>
         (file_path: &Option<P>, city: &str)
         -> Result<Vec<PopulationCount>, CliError> {
    let mut found = vec![];
    let input: Box<io::Read> = match *file_path {
        None => Box::new(io::stdin()),
        Some(ref file_path) => Box::new(try!(File::open(file_pat
h))),
    };
    let mut rdr = csv::Reader::from_reader(input);
    for row in rdr.decode::<Row>() {
        let row = try!(row);
        match row.population {
            None => { } // Skip it.
            Some(count) => if row.city == city {
                found.push(PopulationCount {
                    city: row.city,
                    country: row.country,
                    count: count,
                });
            },
        }
    }
    if found.is_empty() {
        Err(CliError::NotFound)
    } else {
        Ok(found)
    }
}
```

不再需要其他的修改。

增加功能

编写泛型代码是很好的，因为泛用性是很酷的，并且之后会变得很有用。不过有时并不值得这么做。看看我们上一部分我们是怎么做的：

1. 定义了一个新的错误类型。

2. 增加 `Error`，`Display` 和两个 `From` 实现。

这里最大的缺点是我们的程序并没有改进多少。这里仍然有很多用 `enum` 代表错误的额外操作，特别是在这样短小的程序里。

像我们这样使用自定义错误类型的一个有用的方面是 `main` 函数现在可以选择不同的处理错误的方式。之前使用 `Box<Error>` 的时候并没有什么选择：只能打印信息。我们现在仍可以这么做，不过只是在我们想这么做的时候，例如，添加一个 `--quiet` flag？`--quiet` flag 应该能够消除任何冗余的输出。

现在如果程序不能匹配一个城市，它会打印一个信息说它不能。这可能有点蠢，尤其是你想要你的程序能在 shell 脚本中使用的时候。

所以让我们开始增加 flag。就像之前一样，我们需要修改用法字符串，并给选项变量添加 flag。当我们写完这些，Getopts 会搞定剩下的操作：

```
...
    let mut opts = Options::new();
    opts.optopt("f", "file", "Choose an input file, instead of using STDIN.", "NAME");
    opts.optflag("h", "help", "Show this usage message.");
    opts.optflag("q", "quiet", "Silences errors and warnings.");
...
```

现在我们只需要实现我们的“安静”功能。这要求我们修改 `mian` 中的 case analysis：

```
use std::process;
...
    match search(&data_path, city) {
        Err(CliError::NotFound) if matches.opt_present("q") => process::exit(1),
        Err(err) => panic!("{}", err),
        Ok(pops) => for pop in pops {
            println!("{}, {}: {:?}", pop.city, pop.country, pop.count);
        }
    }
...
```

当然，在出现 IO 错误或者数据解析失败时我们并不想变得安静。因此，我们用 case analysis 来检查错误类型是否是 `NotFound` 以及 `--quiet` 是否被启用。如果，搜索失败了，我们仍然使用一个错误码退出（使用 `grep` 的传统）。

如果我们还在用 `Box<Error>` ，那么实现 `--quiet` 功能将变得很复杂。

我们的案例学习讲了很多东西。从这时起，你应该能够在现实生活中编写带有合适错误处理的程序和库了。

精简版

因为这个章节很长，有一个 Rust 错误处理的快速总结是很有帮助的。有很多好的“经验规则”。需要强调的是他们并非教条。这里每一个建议都可能有适当的理由予以反驳！

- 如果你在写小的事例代码这时错误处理显得负担过重，可能使用 `unwrap` （ `Result::unwrap` ， `Option::unwrap` ，或是更可取的 `Option::expect` ）是足够的。你的代码的客户应该知道如何正确的处理错误。（如果他们并不知道，教会他们吧！）
- 如果你在 hack（quick 'n' dirty）程序，不要为你使用 `unwrap` 而感羞愧。不过你被警告过了：如果别人踩到了坑，不要因为他们对糟糕的错误信息火冒三丈而感到惊讶！
- 如果你在 hack 程序并对 panic 感到羞愧，那么使用 `String` 或者 `Box<Error>` 作为你的错误类型。
- 否则，在程序中，定义你自己的错误类型并实现合适的 `From` 和 `Error` 来让 `try!` 宏变得更工程化。
- 如果你在写一个库并且它可能产生错误，定义你自己的错误类型并实现 `std::error::Error` trait。如果可以的话，实现 `From` 来让你的库代码和调用者的代码更加容易编写。（因为 Rust 的一致性规则，调用者不能为你的错误类型实现 `From` ，所以你的库应该实现。）
- 学习定义于 `Option` 和 `Result` 中的组合。只使用他们有时可能比较累人，不过我个人发现合理的结合 `try!` 和组合是比较诱人的。 `and_then` ， `map` 和 `unwrap_or` 是我们的最爱。

选择你的保证

> choosing-your-guarantees.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

Rust 的一个重要特性是允许我们控制一个程序的开销和（安全）保证。

Rust 标准库中有多种“wrapper 类型”的抽象，他们代表了大量在开销，工程学和安全保证之间的权衡。很多让你在运行时和编译时增强之间选择。这一部分将会详细解释一些特定的抽象。

在开始之前，强烈建议你阅读 Rust 的所有权和借用。

基础指针类型

`Box<T>`

Box\是一个“自我拥有的”，或者“装箱”的指针。因为它可以维持引用和包含的数据，它是数据的唯一的拥有者。特别的，当执行类似如下代码时：

```
let x = Box::new(1);
let y = x;
// `x` is no longer accessible here.
```

这里，装箱被移动进了 `y` 。因为 `x` 不再拥有它，此后编译器不再允许程序猿使用 `x` 。相似的一个函数可以通过返回装箱来移出函数。

当一个装箱（还没有被移动的）离开了作用域，析构函数将会运行。这个析构函数负责释放内部的数据。

这是一个动态分配的零开销抽象。如果你想要在堆上分配一些内存并安全的传递这些内存的指针，这是理想的情况。注意你将只能通过正常的借用规则来共享引用，这些在编译时被检查。

`&T` 和 `&mut T`

这分别是不可变和可变引用。他们遵循“读写锁”的模式，也就是你只可能拥有一个数据的可变引用，或者任意数量的不可变引用，但不是两者都有。这个保证在编译时执行，并且没有明显的运行时开销。在大部分情况这两个指针类型有能力在代码块之间廉价的共享引用。

这些指针不能在超出他们的生命周期的情况下被拷贝。

`*const T` 和 `*mut T`

这些是C风格的指针，并没附加生命周期或所有权。他们只是指向一些内存位置，没有其他的限制。他们能提供的唯一的保证是除非在标记为 `unsafe` 的代码中他们不会被解引用。

他们在构建像 `Vec<T>` 这样的安全，低开销抽象时是有用的，不过应该避免在安全代码中使用。

`Rc<T>`

这是第一个我们将会介绍到的有运行时开销的包装类型。

Rc\是一个引用计数指针。换句话说，这让我们拥有相同数据的多个“有所有权”的指针，并且数据在所有指针离开作用域后将被释放（析构函数将会执行）。

在内部，它包含一个共享的“引用计数”（也叫做“refcount”），每次 `Rc` 被拷贝时递增，而每次 `Rc` 离开作用域时递减。 `Rc<T>` 的主要职责是确保共享的数据的析构函数被调用。

这里内部的数据是不可变的，并且如果创建了一个循环引用，数据将会泄露。如果我们想要数据在存在循环引用时不被泄漏，我们需要一个垃圾回收器。

保证

这里（ `Rc<T>` ）提供的主要保证是，直到所有引用离开作用域后，相关数据才会被销毁。

当我们想要动态分配并在程序的不同部分共享一些（只读）数据，且不确定哪部分程序会最后使用这个指针时，我们应该用 `Rc<T>` 。当 `&T` 不可能静态地检查正确性，或者程序员不想浪费开发时间编写反人类的代码时，它可以作为 `&T` 的可行的替代。

这个指针并不是线程安全的，并且Rust也不会允许它被传递或共享给别的线程。这允许你避免在不必要的情况下的原子性开销。

`Rc<T>` 有个姐妹版智能指针类型—— `Weak<T>` 。它是一个既没有所有权、也不能被借用的智能指针。它也比较像 `&T` ，但并没有生命周期的限制--一个 `Weak<T>` 可以一直存活。然而，尝试对其内部数据进行访问可能失败并返回 `None` ，因为它可以比有所有权的 `Rc` 存活更久。这对循环数据结构和一些其他类型是有用的。

开销

随着内存使用增加， `Rc<T>` 是一次性的分配，虽然相比一个常规 `Box<T>` 它会多分配额外两个字（也就是说，两个 `usize` 值）。（"强"引用计数相比"弱"引用计数）。

`Rc<T>` 分别在拷贝和离开作用域时会产生递增/递减引用计数的计算型开销。注意拷贝将不会进行一次深度复制，相反它会简单的递增内部引用计数并返回一个 `Rc<T>` 的拷贝。

Cell 类型

`Cell` 提供内部可变性。换句话说，他们包含的数据可以被修改，即便是这个类型并不能以可变形式获取（例如，当他们位于一个 `&` 指针或 `Rc<T>` 之后时）。

对此 `cell` 模块的文档有一个非常好的解释。

这些类型经常在结构体字段中出现，不过他们也可能在其他一些地方找到。

`Cell<T>`

Cell\是一个提供了零开销内部可变性的类型，不过只用于 `Copy` 类型。因为编译器知道它包含的值对应的所有数据都位于栈上，所以并没有通过简单的替换数据而导致任何位于引用之后的数据泄露（或者更糟！）的担心。

然而使用这个封装仍有可能违反你自己的不可变性，所以谨慎的使用它。它是一个很好的标识，表明一些数据块是可变的并且可能在你第一次读取它和当你想要使用它时的值并不一样。

```rust
use std::cell::Cell;

let x = Cell::new(1);
let y = &x;
let z = &x;
x.set(2);
y.set(3);
z.set(4);
println!("{}", x.get());
```

注意这里我们可以通过多个不可变的引用改变相同的值。

这与如下代码有相同的运行时开销：

```rust
let mut x = 1;
let y = &mut x;
let z = &mut x;
x = 2;
*y = 3;
*z = 4;
println!("{}", x);
```

不过它有额外的优势，它确实能够编译成功。（高级黑？）

保证

这个类型放宽了当没有必要时"没有因可变性导致的混淆"的限制。然而，这也放宽了这个限制提供的保证；所以当你的不可变量依赖存储在 `Cell` 中的数据，你应该多加小心。

这对改变基本类型和其他 `Copy` 类型非常有用，当通过 `&` 和 `&mut` 的静态规则并没有其他简单合适的方法改变他们的值时。

`Cell` 并不让你获取数据的内部引用，它让我们可以自由改变值。

开销

使用 `Cell<T>` 并没有运行时开销，不过你使用它来封装一个很大的（ `Copy` ）结构体，可能更适合封装单独的字段为 `Cell<T>` 因为每次写入都会是一个结构体的完整拷贝。

RefCell<T>

RefCell\也提供了内部可变性，不过并不限制为 `Copy` 类型。

相对的，它有运行时开销。 `RefCell<T>` 在运行时使用了读写锁模式，不像 `&T` / `&mut T` 那样在编译时执行。这通过 `borrow()` 和 `borrow_mut()` 函数来实现，它修改一个内部引用计数并分别返回可以不可变的和可变的解引用的智能指针。当智能指针离开作用域引用计数将被恢复。通过这个系统，我们可以动态的确保当有一个有效的可变借用时绝不会有任何其他有效的借用。如果程序猿尝试创建一个这样的借用，线程将会恐慌。

```
use std::cell::RefCell;

let x = RefCell::new(vec![1,2,3,4]);
{
    println!("{:?}", *x.borrow())
}

{
    let mut my_ref = x.borrow_mut();
    my_ref.push(1);
}
```

与 `Cell` 相似，它主要用于难以或不可能满足借用检查的情况。大体上我们知道这样的改变不会发生在一个嵌套的形式中，不过检查一下是有好处的。

对于大型的，复杂的程序，把一些东西放入 `RefCell` 来将事情变简单是有用的。例如，Rust 编译器内部的 `ctxt` 结构体中的很多 map 都在这个封装中。他们只会在创建时被修改一次（但并不是正好在初始化后），或者在明显分开的地方多次多次修改。然而，因为这个结构体被广泛的用于各个地方，有效的组织可变和不可变的指针将会是困难的（也许是不可能的），并且可能产生大量的难以扩展的 `&` 指针。换句话说， `RefCell` 提供了一个廉价（并不是零开销）的方式来访问它。之后，如果有人增加一些代码来尝试修改一个已经被借用的 cell 时，这将会产生（通常是决定性的）一个恐慌，并会被追溯到那个可恶的借用上。

相似的，在 Servo 的 DOM 中有很多可变量，大部分对于一个 DOM 类型都是本地的，不过有一些交错在 DOM 中并修改了很多内容。使用 `RefCell` 和 `Cell` 来保护所有的变化可以让我们免于担心到处都是的可变性，并且同时也表明了何处正在发生变化。

注意如果是一个能用 `&` 指针的非常简单的情形应该避免使用 `RefCell` 。

保证

`RefCell` 放宽了避免混淆的改变的静态限制，并代之以一个动态限制。保证本身并没有改变。

开销

`RefCell` 并不分配空间，不过它连同数据还包含一个额外的“借用状态”指示器（一个字的大小）。

在运行时每次借用产生一次引用计数的修改/检查。

同步类型（Synchronous types）

上面的很多类型不能以一种线程安全的方式使用。特别是 `Rc<T>` 和 `RefCell<T>` ，他们都使用非原子的引用计数（原子引用计数可以在不引起数据竞争的情况下在多个线程中递增），不能在多线程中使用。这让他们使用起来更廉价，不过我们也需要这两个类型的线程安全版本。他们以 `Arc<T>` 和 `Mutex<T>` / `RWLock<T>` 的形式存在。

注意非线程安全的类型不能在线程间传递，并且这是在编译时检查的。

`Arc<T>`

Arc\就是一个使用原子引用计数版本的 `Rc<T>` （*Atomic reference count*，因此是“Arc”）。它可以在线程间自由的传递。

C++的 `shared_ptr` 与 `Arc` 类似，然而C++的情况中它的内部数据总是可以改变的。为了语义上与C++的形式相似，我们应该使用 `Arc<Mutex<T>>` ， `Arc<RwLock<T>>` ，或者 `Arc<UnsafeCell<T>>` [1]。最后一个应该只被用在我们能确定使用它并不会造成内存不安全性的情况下。记住写入

一个结构体不是一个原子操作，并且很多像 `vec.push()` 这样的函数可以在内部重新分配内存并产生不安全的行为，所以即便是单一环境也不足以证明 `UnsafeCell` 是安全的。

保证

类似 `Rc` ，它提供了当最后的 `Arc` 离开作用域时（不包含任何的循环引用）其内部数据的析构函数将被执行的（线程安全的）保证。

开销

使用原子引用计数有额外的开销（无论是被拷贝或者离开作用域时都会发生）。当在一个单独的线程中通过一个 `Arc` 共享数据时，任何时候都更倾向于使用 `&` 指针。

`Mutex<T>` 和 `RwLock<T>`

Mutex\和RwLock\通过 RAII guard（guard 是一类直到析构函数被调用时能保持一些状态的对象）提供了互斥功能。对于这两个类型，mutex 直到我们调用 `lock()` 之前它都是无效的，此时直到我们获取锁这个线程都会被阻塞，同时它会返回一个 guard。这个 guard 可以被用来访问它的内部数据（可变的），而当 guard 离开作用域锁将被释放。

```
{
    let guard = mutex.lock();
    // `guard` dereferences mutably to the inner type.
    *guard += 1;
} // Lock is released when destructor runs.
```

`RwLock` 对多线程读有额外的效率优势。只要没有 writer，对于共享的数据总是可以安全的拥有多个 reader；同时 `RwLock` 让reader们获取一个“读取锁”。这样的锁可以并发的获取并通过引用计数记录。writer 必须获取一个“写入锁”，它只有在所有 reader 都离开作用域时才能获取。

保证

这两个类型都提供了线程间安全的共享可变性，然而他们易于产生死锁。一些额外的协议层次的安全性可以通过类型系统获取。

开销

他们在内部使用类原子类型来维持锁，这样的开销非常大（他们可以阻塞处理器所有的内存读取知道他们执行完毕）。而当有很多并发访问时等待这些锁也将是很慢的。

组合（Composition）

阅读 Rust 代码时的一个常见的痛苦之处是遇到形如 `Rc<RefCell<Vec<T>>>` 这样的类型（或者诸如此类的更复杂的组合）。这些组合式干什么的，和为什么作者会选这么一个类型（以及何时你应该在自己的代码中使用这样一个类型）的理由并不总是显而易见的。

通常，将你需要的保证组合到一起是一个例子，而不为无关紧要的东西产生开销。

例如，`Rc<RefCell<T>>` 就是一个这样的组合。`Rc<T>` 自身并不能可变的解引用；因为 `Rc<T>` 可以共享，而共享的可变性可以导致不安全的行为，所以我们在其中放入 `RefCell<T>` 来获得可以动态验证的共享可变性。现在我们有了共享的可变数据，不过它只能以只有一个 writer（没有 reader）或多个 reader 的方式共享。

现在，我们可以更进一步，并拥有 `Rc<RefCell<Vec<T>>>` 或 `Rc<Vec<RefCell<T>>>`，他们都是可共享可改变的 vector，不过他们并不一样。

前者，`RefCell<T>` 封装了 `Vec<T>`，所以 `Vec<T>` 整体是可变的。与此同时，同一时刻只能有一个整个 `Vec` 的可变借用。这意味着你的代码不能同时通过不同的 `Rc` 句柄来操作 vector 的不同元素。然而，我们可以随意的从 `Vec<T>` 中加入或取出元素。这类似于一个有运行时借用检查的 `&mut Vec<T>`。

后者，借用作用于单独的元素，不过vector整体是不可变的。因此，我们可以独立的借用不同的元素，不过我们对vector加入或取出元素。这类似于 `&mut [T]`[2]，不过同样会在运行时做借用检查。

在并发程序中，我们有一个使用 `Arc<Mutex<T>>` 的类似场景，它提供了共享可变性和所有权。

当阅读使用这些类型的代码时，一步步的阅读并关注他们提供的保证/开销。

当选择一个组合类型的时候，我们必须反过来思考；搞清楚我们需要何种保证，以及在组合中的何处我们需要他们。例如，如果面对一个 `Vec<RefCell<T>>` 和 `RefCell<Vec<T>>` 之间的选择，我们需要明确像上面讲到的那样的权衡并选择其一。

---

1. `Arc<UnsafeCell<T>>` 实际上并不能编译因为 `UnsafeCell<T>` 并不是 `Send` 或 `Sync` 的，不过我们可以把它 wrap 进一个类型并且手动为其实现 `Send` / `Sync` 来获得 `Arc<Wrapper<T>>`，它的 `Wrapper` 是 `struct Wrapper<T>(UnsafeCell<T>)`。↩

2. `&[T]` 和 `&mut [T]` 是切片（slice）；他们包含一个指针和一个长度并可以引用一个vector或数组的一部分。`&mut [T]` 能够改变它的元素，不过长度不能改变。↩

外部函数接口(FFI)

> ffi.md
> commit 59f76f1a2919641f246da62751eee620b4736acf

介绍

本教程会使用snappy压缩/解压缩库来作为一个 Rust 编写外部语言代码绑定的介绍。目前 Rust 还不能直接调用 C++ 库，不过 snappy 库包含一个 C 接口（记录在snappy-c.h中）。

一个关于 libc 的说明

很多这些例子使用 `libc` crate，它提供了很多 C 类型的类型定义，还有很多其他东西。如果你正在自己尝试这些例子，你会需要在你的 `Cargo.toml` 中添加 `libc`：

```
[dependencies]
libc = "0.2.0"
```

并在你的 crate 根文件添加 `extern crate libc;`

调用外部函数

下面是一个最简单的调用其它语言函数的例子，如果你安装了snappy的话它将能够编译：

```
# #![feature(libc)]
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> si
ze_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x
);
}
```

`extern` 块是一个外部库函数标记的列表，在这里例子中是 C ABI。 `#[link(...)]` 属性用来指示链接器链接snappy 库来解析符号。

外部函数被假定为不安全的所以调用它们需要包装在 `unsafe {}` 中，用来向编译器保证大括号中代码是安全的。C 库经常提供不是线程安全的接口，并且几乎所有以指针作为参数的函数不是对所有输入时有效的，因为指针可以是垂悬的，而且裸指针超出了 Rust 安全内存模型的范围。

当声明外部语言的函数参数时，Rust 编译器不能检查它是否正确，所以指定正确的类型是保证绑定运行时正常工作的一部分。

`extern` 块可以扩展以包括整个 snappy API：

```
# #![feature(libc)]
extern crate libc;
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_
int;
    fn snappy_max_compressed_length(source_length: size_t) -> si
ze_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size
_t) -> c_int;
}
# fn main() {}
```

创建安全接口

原始 C API 需要需要封装才能提供内存安全性和利用像向量这样的高级内容。一个库可以选择只暴露出安全的，高级的接口并隐藏不安全的底层细节。

包装用到了缓冲区的函数涉及使用 `slice::raw` 模块来将Rust向量作为内存指针来操作。Rust 的 vector 确保是一个连续的内存块。它的长度是当前包含的元素个数，而容量则是分配内存的大小。长度小于或等于容量。

```
# #![feature(libc)]
# extern crate libc;
# use libc::{c_int, size_t};
# unsafe fn snappy_validate_compressed_buffer(_: *const u8, _: s
ize_t) -> c_int { 0 }
# fn main() {}
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len(
) as size_t) == 0
    }
}
```

上面的 `validate_compressed_buffer` 封装使用了一个 `unsafe` 块，不过它通过从函数标记汇总去掉 `unsafe` 从而保证了对于所有输入调用都是安全的。

`snappy_compress` 和 `snappy_uncompress` 函数更复杂，因为输出也使用了被分配的缓冲区。

`snappy_max_compressed_length` 函数可以用来分配一个所需最大容量的向量来存放压缩的输出。接着这个向量可以作为一个输出参数传递给 `snappy_compress` 。另一个输出参数也被传递进去并设置了长度，可以用它来获取压缩后的真实长度。

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_compress(a: *const u8, b: size_t, c: *mut u8,
#                           d: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_max_compressed_length(a: size_t) -> size_t {
a }
# fn main() {}
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as usize);
        dst
    }
}
```

解压是相似的，因为 snappy 储存了未压缩的大小作为压缩格式的一部分并且 `snappy_uncompressed_length` 可以取得所需缓冲区的实际大小。

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_uncompress(compressed: *const u8,
#                              compressed_length: size_t,
#                              uncompressed: *mut u8,
#                              uncompressed_length: *mut size_t)
-> c_int { 0 }
# unsafe fn snappy_uncompressed_length(compressed: *const u8,
#                                       compressed_length: size_t
,
#                                       result: *mut size_t) -> c
_int { 0 }
# fn main() {}
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as usize);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) ==
0 {
            dst.set_len(dstlen as usize);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

接下来，我们可以添加一些测试来展示如何使用他们：

```
# #![feature(libc)]
```

```
# extern crate libc;
# use libc::{c_int, size_t};
# unsafe fn snappy_compress(input: *const u8,
#                           input_length: size_t,
#                           compressed: *mut u8,
#                           compressed_length: *mut size_t)
#                           -> c_int { 0 }
# unsafe fn snappy_uncompress(compressed: *const u8,
#                             compressed_length: size_t,
#                             uncompressed: *mut u8,
#                             uncompressed_length: *mut size_t)
#                             -> c_int { 0 }
# unsafe fn snappy_max_compressed_length(source_length: size_t)
-> size_t { 0 }
# unsafe fn snappy_uncompressed_length(compressed: *const u8,
#                                      compressed_length: size_t
,
#                                      result: *mut size_t)
#                                      -> c_int { 0 }
# unsafe fn snappy_validate_compressed_buffer(compressed: *const
u8,
#                                             compressed_length:
 size_t)
#                                             -> c_int { 0 }
# fn main() { }

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn valid() {
        let d = vec![0xde, 0xad, 0xd0, 0x0d];
        let c: &[u8] = &compress(&d);
        assert!(validate_compressed_buffer(c));
        assert!(uncompress(c) == Some(d));
    }

    #[test]
    fn invalid() {
```

```
        let d = vec![0, 0, 0, 0];
        assert!(!validate_compressed_buffer(&d));
        assert!(uncompress(&d).is_none());
    }

    #[test]
    fn empty() {
        let d = vec![];
        assert!(!validate_compressed_buffer(&d));
        assert!(uncompress(&d).is_none());
        let c = compress(&d);
        assert!(validate_compressed_buffer(&c));
        assert!(uncompress(&c) == Some(d));
    }
}
```

析构函数

外部库经常把资源的所有权传递给调用函数。当这发生时，我们必须使用 Rust 析构函数来提供安全性和确保释放了这些资源（特别是在恐慌的时候）。

关于析构函数的更多细节，请看 `Drop` trait

在 Rust 函数中处理 C 回调（Callbacks from C code to Rust functions）

一些外部库要求使用回调来向调用者反馈它们的当前状态或者即时数据。可以传递在 Rust 中定义的函数到外部库中。要求是这个回调函数被标记为 `extern` 并使用正确的调用约定来确保它可以在 C 代码中被调用。

接着回调函数可以通过一个C库的注册调用传递并在后面被执行。

一个基础的例子：

Rust代码：

```
extern fn callback(a: i32) {
    println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
        register_callback(callback);
        trigger_callback(); // Triggers the callback.
    }
}
```

C 代码：

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;

int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // Will call callback(7) in Rust.
}
```

这个例子中 Rust 的 `main()` 会调用 C 中的 `trigger_callback()`，它会反过来调用 Rust 中的 `callback()`。

在 Rust 对象上使用回调（Targeting callbacks to Rust objects）

之前的例子展示了一个全局函数是如何在C代码中被调用的。然而我们经常希望回调是针对一个特殊 Rust 对象的。这个对象可能代表对应 C 语言中的封装。

这可以通过向 C 库传递这个对象的不安全指针来做到。C 库则可以根据这个这个通知中的指针来取得 Rust 对象。这允许回调不安全的访问被引用的 Rust 对象。

Rust 代码：

```
#[repr(C)]
struct RustObject {
    a: i32,
    // Other members...
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // Update the value in RustObject with the value receive
d from the callback:
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) ->
i32;
   fn trigger_callback();
}

fn main() {
    // Create the object that will be referenced in the callback:

    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C 代码：

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback c
allback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Will call callback(&rustObject, 7) in Rus
t.
}
```

异步回调

在之前给出的例子中回调在一个外部 C 库的函数调用后直接就执行了。在回调的执行过程中当前线程控制权从 Rust 传到了 C 又传到了 Rust，不过最终回调和和触发它的函数都在一个线程中执行。

当外部库生成了自己的线程并触发回调时情况就变得复杂了。在这种情况下回调中对 Rust 数据结构的访问时特别不安全的并必须有合适的同步机制。除了像互斥量这种经典同步机制外，另一种可能就是使用通道（ `std::sync::mpsc` ）来从触发回调的 C 线程转发数据到 Rust 线程。

如果一个异步回调指定了一个在 Rust 地址空间的特殊 Rust 对象，那么在确保在对应 Rust 对象被销毁后不会再有回调被 C 库触发就格外重要了。这一点可以通过在对象的析构函数中注销回调和设计库使其确保在回调被注销后不会再被触发来取得。

链接

在 `extern` 上的 `link` 属性提供了基本的构建块来指示 `rustc` 如何连接到原生库。现在有两种被接受的链接属性形式：

* `#[link(name = "foo")]`
* `#[link(name = "foo", kind = "bar")]`

在这两种形式中，`foo` 是我们链接的原生库的名字，而在第二个形式中 `bar` 是编译器要链接的原生库的类型。目前有 3 种已知的原生库类型：

* 动态 - `#[link(name = "readline")]`
* 静态 - `#[link(name = "my_build_dependency", kind = "static")]`
* Frameworks - `#[link(name = "CoreFoundation", kind = "framework")]`

注意 Frameworks 只支持 OSX 平台。

不同 `kind` 的值意味着链接过程中不同原生库的参与方式。从链接的角度看，rus t 编译器创建了两种组件：部分的（rlib/staticlib）和最终的（dylib/binary）。原生动态库和框架会从扩展到最终组件部分，而静态库则完全不会扩展。

一些关于这些模型如何使用的例子：

* 一个原生构建依赖。有时编写部分Rust代码时需要一些C/C++代码，另外使用发行为库格式的C/C++代码只是一个负担。在这种情况下，代码会被归档为 `libfoo.a` 然后rust包装箱可以通过 `#[link(name = "foo", kind = "static")]` 声明一个依赖。

  不管包装箱输出为何种形式，原生静态库将会包含在输出中，这意味着分配一个原生静态库是没有必要的。

* 一个正常动态库依赖。通用系统库（像 `readline` ）在大量系统上可用，通常你找不到这类库的静态拷贝。当这种依赖被添加到包装箱里时，部分目标（比如rlibs）将不会链接这些库，但是当rlib被包含进最终目标（比如二进制文件）时，原生库将被链接。

在OSX上，框架与动态库有相同的语义。

不安全块（Unsafe blocks）

一些操作，像解引用不安全的指针或者被标记为不安全的函数只允许在 unsafe 块中使用。unsafe 块隔离的不安全性并向编译器保证不安全代码不会泄露到块之外。

不安全函数，另一方面，将它公布于众。一个不安全的函数这样写：

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

这个函数只能被从 `unsafe` 块中或者 `unsafe` 函数调用。

访问外部全局变量（Accessing foreign globals）

外部 API 经常导出一个全局变量来进行像记录全局状态这样的工作。为了访问这些变量，你可以在 `extern` 块中用 `static` 关键字声明它们：

```
# #![feature(libc)]
extern crate libc;

#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("You have readline version {} installed.",
             unsafe { rl_readline_version as i32 });
}
```

另外，你可能想修改外部结接口提供的全局状态。为了做到这一点，声明为 `mut` 这样我们就可以改变它了。

```
# #![feature(libc)]
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

        println!("{:?}", rl_prompt);

        rl_prompt = ptr::null();
    }
}
```

注意与 `static mut` 变量的所有交互都是不安全的，包括读或写。与全局可变量打交道需要足够的注意。

外部调用约定（Foreign calling conventions）

大部分外部代码导出为一个 C 的 ABI，并且 Rust 默认使用平台 C 的调用约定来调用外部函数。一些外部函数，尤其是大部分 Windows API，使用其它的调用约定。Rust 提供了一个告诉编译器应该用哪种调用约定的方法：

```
# #![feature(libc)]
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> li
bc::c_int;
}
# fn main() { }
```

这适用于整个 `extern` 块。被支持的ABI约束的列表为：

- `stdcall`
- `aapcs`
- `cdecl`
- `fastcall`
- `vectorcall` 目前隐藏于 `abi_vectorcall` gate 之后并倾向于改变。
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`
- `sysv64`

列表中大部分 ABI 都是自解释的，不过 `system` ABI 可能看起来有点奇怪。这个约束会选择任何能和目标库正确交互的 ABI。例如，在x86架构上，这意味着会使用 `stdcall` ABI。然而，在 x86_64 上 windows 使用 `C` 调用约定，所以 `C` 会被使用。这意味在我们之前的例子中，我们可以使用 `extern "system" { ... }` 定义一个适用于所有 windows 系统的块，而不仅仅是 x86 系统。

外部代码交互性（Interoperability with foreign code）

只有当 `#[repr(C)]` 属性被用于结构体时Rust能确保 `struct` 的布局兼容平台的 C 的表现。`#[repr(C, packed)]` 可以用来不对齐的排列结构体成员。`#[repr(C)]` 也可以被用于一个枚举。

Rust 拥有的装箱（`Box<T>`）使用非空指针作为指向他包含的对象的句柄。然而，它们不应该手动创建因为它们由内部分分配器托管。引用可以被安全的假设为直接指向数据的非空指针。然而，打破借用检查和可变性规则并不能保证安全，所以倾向于只在需要时使用裸指针（`*`）因为编译器不能为它们做更多假设。

向量和字符串共享同样基础的内存布局，`vec` 和 `str` 模块中可用的功能可以操作 C API。然而，字符串不是 `\0` 结尾的。如果你需要一个NUL结尾的字符串来与 C 交互，你需要使用 `std::ffi` 模块中的 `CString` 类型。

标准库中的 `libc` 模块包含类型别名和 C 标准库中的函数定义，Rust 默认链接 `libc` 和 `libm` 。

可变参数函数

在 C 语言中，函数是“可变参数的”，这意味着它可以接受可变数量的参数。在 Rust 中可以通过在外部函数声明中指定 `...` 来实现：

```rust
extern {
    fn foo(x: i32, ...);
}

fn main() {
    unsafe {
        foo(10, 20, 30, 40, 50);
    }
}
```

普通的 Rust 不能是可变参数的：

```rust
// This will not compile

fn foo(x: i32, ...) { }
```

“可空指针优化”（The "nullable pointer optimization"）

特定的 Rust 类型被定义为永远不能为 `null` 。这包括引用（ `&T` ， `&mut T` ），装箱（ `Box<T>` ），和函数指针（ `extern "abi" fn()` ）。当调用 C 接口时，可以为 `null` 的指针被广泛使用，这好像会需要使用到一些混乱的 `transmutes` 和/或不安全代码来处理与 Rust 类型间的相互转换。不过，Rust 语言提供了一个变通方案。

作为一个特殊的例子，一类只包含两个 variant 的 `enum` 适合用来进行“可空指针优化”，其中一个 variant 不包含数据，另一个包含一个上述不可空类型的字段。这意味着不需要额外的空间来进行判别，相反，空的 variant 表现为将一个 `null` 值放入不可空的字段。这也被称为一种优化，不过不同于别读优化它保证适用于合适的类型。

得益于这种可空指针优化的最常见的例子是 `Option<T>` ，这里 `None` 对应 `null` 。所以 `Option<extern "C" fn(c_int) -> c_int>` 是一个用于 C ABI 的可空函数指针的正确方式（对应 C 类型 `int (*)(int)` ）。

这是一个不太自然的例子。假如一些 C 库拥有一个注册回调的功能，它在特定情况下被调用。回调传递了一个函数指针和一个整型并应该以这个整型作为参数执行这个函数。所以我们有一些双向穿越 FFI boundary 的函数指针。

```
# #![feature(libc)]
extern crate libc;
use libc::c_int;

# #[cfg(hidden)]
extern "C" {
    /// Registers the callback.
    fn register(cb: Option<extern "C" fn(Option<extern "C" fn(c_
int) -> c_int>, c_int) -> c_int>);
}
# unsafe fn register(_: Option<extern "C" fn(Option<extern "C" fn
(c_int) -> c_int>,
#                                            c_int) -> c_int>)
# {}

/// This fairly useless function receives a function pointer and
 an integer
/// from C, and returns the result of calling the function with
the integer.
/// In case no function is provided, it squares the integer by d
efault.
extern "C" fn apply(process: Option<extern "C" fn(c_int) -> c_in
t>, int: c_int) -> c_int {
    match process {
        Some(f) => f(int),
        None    => int * int
    }
}

fn main() {
    unsafe {
        register(Some(apply));
    }
}
```

而 C 代码看起来像这样：

```
void register(void (*f)(void (*)(int), int)) {
    ...
}
```

并不需要 `transmute` ！

在 C 中调用 Rust 代码

你可能会希望这么编译 Rust 代码以便可以在 C 中调用。这是很简单的，不过需要一些东西：

```
#[no_mangle]
pub extern fn hello_rust() -> *const u8 {
    "Hello, world!\0".as_ptr()
}
# fn main() {}
```

`extern` 使这个函数遵循 C 调用约定，就像之前讨论外部调用约定时一样。 `no_mangle` 属性关闭Rust的命名改编，这样它更容易链接。

FFI 和 panic

当使用 FFI 时留意 `panic!` 是很重要的。一个跨越FFI边界的 `panic!` 是未定义行为。如果你在编写可能 panic 的代码，你应该使用 `catch_unwind()` 在一个闭包中运行它：

```
use std::panic::catch_unwind;

#[no_mangle]
pub extern fn oh_no() -> i32 {
    let result = catch_unwind(|| {
        panic!("Oops!");
    });

    match result {
        Ok(_) => 0,
        Err(_) => 1,
    }
}
fn main() {}
```

注意 `catch_unwind()` 只捕获那些不严重的（unwinding）panic，不是那些会终止进程的。查看 `catch_unwind()` 的文档来获取更多信息。

表示 opaque 结构体

有时一个 C 库想要提供某种指针，不过并不想让你知道它需要的内部细节。最简单的方法是使用一个 `void *` 参数：

```
void foo(void *arg);
void bar(void *arg);
```

我们可以使用 `c_void` 在 Rust 中表示它：

```
# #![feature(libc)]
extern crate libc;

extern "C" {
    pub fn foo(arg: *mut libc::c_void);
    pub fn bar(arg: *mut libc::c_void);
}
# fn main() {}
```

这是处理这种情形完美有效的方式。然而，我们可以做的更好一点。为此，一些 C 库会创建一个 `struct` ，结构体的细节和内存布局是私有的。这提供了一些类型安全性。这种结构体叫做 `opaque` 。这是一个 C 的例子：

```
struct Foo; /* Foo is a structure, but its contents are not part
 of the public interface */
struct Bar;
void foo(struct Foo *arg);
void bar(struct Bar *arg);
```

在 Rust 中，让我们用 `enum` 创建自己的 opaque 类型：

```
pub enum Foo {}
pub enum Bar {}

extern "C" {
    pub fn foo(arg: *mut Foo);
    pub fn bar(arg: *mut Bar);
}
# fn main() {}
```

通过一个没有变量的 `enum` ，我们创建了一个不能实例化的 opaque 类型，因为它没有变量。不过因为我们的 `Foo` 和 `Bar` 是不同类型，我们可以安全的获取这两个类型，所以我们不可能不小心向 `bar()` 传递一个 `Foo` 的指针。

`Borrow` 和 `AsRef`

> borrow-and-asref.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

`Borrow` 和 `AsRef` 特性非常相似。这是一个快速的关于这两个特性意义的复习。

`Borrow`

`Borrow` 特性用于当你处于某种目的写了一个数据结构，并且你想要使用一个要么拥有要么借用的类型作为它的同义词。

例如，`HashMap` 有一个用了 `Borrow` 的 `get` 方法：

```
fn get<Q: ?Sized>(&self, k: &Q) -> Option<&V>
    where K: Borrow<Q>,
          Q: Hash + Eq
```

这个签名非常复杂。 `k` 参数是我们感兴趣的。它引用了一个 `HashMap` 自身的参数：

```
struct HashMap<K, V, S = RandomState> {
```

`k` 参数是 `HashMap` 用的 `key` 类型。所以，再一次查看 `get()` 的签名，我们可以在键实现了 `Borrow<Q>` 时使用 `get()` 。这样，我们可以创建一个 `HashMap` ，它使用 `String` 键，不过在我们搜索时使用 `&str` ：

```
use std::collections::HashMap;

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

这是因为标准库中有 `impl Borrow<str> for String` （为 String 实现了 Borrow）。

对于多数类型，当你想要获取一个自我拥有或借用的类型， `&T` 就足够了。不过当有多于一种借用的值时， `Borrow` 就能起作用了。引用和 `slice` 就是一个能体现这一点的地方：你可以有 `&[T]` 或者 `&mut [T]` 。如果我们想接受这两种类型， `Borrow` 就是你需要的：

```
use std::borrow::Borrow;
use std::fmt::Display;

fn foo<T: Borrow<i32> + Display>(a: T) {
    println!("a is borrowed: {}", a);
}

let mut i = 5;

foo(&i);
foo(&mut i);
```

这会打印出 `a is borrowed: 5` 两次。

AsRef

`AsRef` 特性是一个转换特性。它用来在泛型中把一些值转换为引用。像这样：

```
let s = "Hello".to_string();

fn foo<T: AsRef<str>>(s: T) {
    let slice = s.as_ref();
}
```

我应该用哪个？

我们可以看到它们有些相似：它们都处理一些类型的自我拥有和借用版本。然而，它们还是有些不同。

选择 `Borrow` 当你想要抽象不同类型的借用，或者当你创建一个数据结构它把自我拥有和借用的值看作等同的，例如哈希和比较。

选择 `AsRef` 当你想要直接把一些值转换为引用，和当你在写泛型代码的时候。

发布途径

> release-channels.md
> commit ccb1d87d6faa9ff528d22b96595a0e2cbb16c0f2

Rust 项目使用一个叫做“发布途径”的概念来管理发布。理解这个选择你的项目应该使用哪个版本的 Rust 的过程是很重要的。

概览

Rust 发布有 3 种途径：

- 开发版（Nightly）
- 测试版（Beta）
- 稳定版（Stable）

新的开发发布每天创建一次。每 6 个星期，最后的开发版被提升为“测试版”。在这时，它将只会收到修改重大错误的补丁。6 个星期之后，测试版被提升为“稳定版”，而成为下一个版本的 `1.x` 发布。

这个过程并行发生。所以每 6 个星期，在同一天，开发变测试，测试变稳定。当 `1.x` 发布时的同时， `1.(x + 1)-beta` 被发布，而开发版变为第一版的 `1.(x + 2)-nightly` 。

选择一个版本

通常来说，除非你有一个特定的原因，你应该使用稳定发布途径。这个发布意为用于普通用户。

然而，根据你对Rust的兴趣，你可能会选择使用开发构建。基本的权衡是：在开发途径，你可以使用不稳定的，新的Rust功能。然而，不稳定功能倾向于改变，所以任何新的开发版发布可能会破坏你的代码。如果你使用稳定发布，你不能使用实验功能，不过下一个Rust发布将不会因为破坏性改变造成显著的问题。

通过持续集成（CI）改善生态系统

那么测试版怎么样呢？我们鼓励所有使用稳定发布途径的 Rust 用户在他们的持续集成系统中也针对测试途径进行测试。这会帮助警告团队以防出现一个意外的退步（regression）。

另外，针对开发版测试能够更快的捕获退步，因此如果你不介意一个第三种构建（环境），我们也会感激你针对开发版进行测试。

作为一个例子，很多 Rust 程序猿使用Travis来测试他们的 crate，这是一个开源的免费项目。Travis直接支持Rust，并且你可以用类似这样的一个 `.travis.yml` 文件来测试所有的版本：

```
language: rust
rust:
  - nightly
  - beta
  - stable

matrix:
  allow_failures:
    - rust: nightly
```

通过这个配置，Travis 将会测试所有三个版本，不过如果有什么东西在开发版中失败了，你的构建将不会失败。建议你在任何 CI 系统中使用类似的配置，查看你正在使用的 CI 系统的文档来获取更多细节。

不使用标准库开发 Rust

> no-stdlib.md
>
> commit 98b18fabddaa942c6bbbb44509b7c00f4b07915f

Rust 的标准库提供了很多有用的功能，不过它假设它的 host 系统的多种功能的支持：线程，网络，堆分配和其他功能。有些系统并没有这些功能，不过，Rust也能在这些系统上工作。为此，我们可以通过一个属性来告诉 Rust 我们不想使用标准库：`#![no_std]`。

> 注意：这个功能技术上是稳定的，不过有些附加条件。其一，你可以构建一个稳定的`#![no_std]`库，但二进制文件不行。关于没有标准库的库文件的细节，查看关于 `#![no_std]` 的章节。

为了使用`#![no_std]`，在 crate 的根文件上加入：

```
#![no_std]

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

很多暴露于标准库中的功能通过`core` crate也同样可用。当我们使用标准库时，Rust 自动将`std`引入到作用域中，允许我们不用显示导入就能使用相关功能。相似的，当使用`#![no_std]`，Rust 会将`core`引入作用域中，以及它的 prelude。这意味着很多代码也是能正常运行的：

```
#![no_std]

fn may_fail(failure: bool) -> Result<(), &'static str> {
    if failure {
        Err("this didn’t work!")
    } else {
        Ok(())
    }
}
```

过程宏（和自定义导出）

> procedural-macros.md
> commit 399708cf7bb12f593ad4bb403e94466d146e743e

在本书接下来的部分，你将看到 Rust 提供了一个叫做“导出（derive）”的机制来轻松的实现 trait。例如，

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}
```

is a lot simpler than

```
struct Point {
    x: i32,
    y: i32,
}

use std::fmt;

impl fmt::Debug for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Point {{ x: {}, y: {} }}", self.x, self.y)
    }
}
```

Rust 包含很多可以导出的 trait，不过也允许定义你自己的 trait。我们可以通过一个叫做“过程宏”的 Rust 功能来实现这个效果。最终，过程宏将会允许 Rust 所有类型的高级元编程，不过现在只能自定义导出。

Hello World

首先需要做的就是为我们的项目新建一个 crate。

```
$ cargo new --bin hello-world
```

我们想要实现的就是能够在导出的类型上调用 `hello_world()` 。就想这样：

```
#[derive(HelloWorld)]
struct Pancakes;

fn main() {
    Pancakes::hello_world();
}
```

再来一些给力的输出，比如“Hello, World! 我叫煎饼（←_←）。”

继续并从用户的角度编写我们的宏。在 `src/main.rs` 中：

```
#[macro_use]
extern crate hello_world_derive;

trait HelloWorld {
    fn hello_world();
}

#[derive(HelloWorld)]
struct FrenchToast;

#[derive(HelloWorld)]
struct Waffles;

fn main() {
    FrenchToast::hello_world();
    Waffles::hello_world();
}
```

好的。现在我们只需实际编写我们的过程宏。目前，过程宏需要位于它自己的 crate 中。最终这个限制会解除，不过现在是必须的。为此，有一个惯例是，对于一个叫 `foo` 的 crate，一个自定义的过程宏叫做 `foo-derive` 。让我们在 `hello-`

`world` 项目中新建一个叫做 `hello-world-derive` 的 crate。

```
$ cargo new hello-world-derive
```

为了确保 `hello-world` crate 能够找到这个新创建的 crate 我们把它加入到项目 toml 文件中：

```
[dependencies]
hello-world-derive = { path = "hello-world-derive" }
```

这里是一个 `hello-world-derive` crate 源码的例子：

```
extern crate proc_macro;
extern crate syn;
#[macro_use]
extern crate quote;

use proc_macro::TokenStream;

#[proc_macro_derive(HelloWorld)]
pub fn hello_world(input: TokenStream) -> TokenStream {
    // Construct a string representation of the type definition
    let s = input.to_string();

    // Parse the string representation
    let ast = syn::parse_derive_input(&s).unwrap();

    // Build the impl
    let gen = impl_hello_world(&ast);

    // Return the generated impl
    gen.parse().unwrap()
}
```

这里有很多内容。我们引入了两个新的 crate：`syn` 和 `quote` 。你可能注意到了，`input: TokenSteam` 直接就被转换成了一个 `String` 。这个字符串是我们要导出的 `HelloWorld` Rust 代码的字符串形式。现在，能对 `TokenStream` 做的唯

一的事情就是把它转换为一个字符串。将来会有更丰富的 API。

所以我们真正需要做的是能够把 Rust 代码解析成有用的东西。这正是 `syn` 出场机会。`syn` 是一个解析 Rust 代码的 crate。我们引入的另外一个 crate 是 `quote` 。它本质上与 `syn` 是成双成对的，因为它可以轻松的生成 Rust 代码。也可以自己编写这些功能，不过使用这些库会更加轻松。编写一个完整 Rust 代码解析器可不是一个简单的工作。

这些代码注释提供了我们总体策略的很好的解释。我们将为导出的类型提供一个 `String` 类型的 Rust 代码，用 `syn` 解析它，（使用 `quote` ）构建 `hello_world` 的实现，接着把它传递回给 Rust 编译器。

最后一个要点：这里有一些 `unwrap()` ，如果你要为过程宏提供一个错误，那么你需要 `panic!` 并提供错误信息。这里，我们从简实现。

好的，让我们编写 `impl_hello_world(&ast)` 。

```
fn impl_hello_world(ast: &syn::MacroInput) -> quote::Tokens {
    let name = &ast.ident;
    quote! {
        impl HelloWorld for #name {
            fn hello_world() {
                println!("Hello, World! My name is {}", stringif
y!(#name));
            }
        }
    }
}
```

这里就是 `quote` 出场的地方。 `ast` 参数是一个代表我们类型（可以是一个 `struct` 或 `enum` ）的结构体。查看文档。这里有一些有用的信息。我们可以通过 `ast.ident` 获取类型的信息。 `quote!` 宏允许我们编写想要返回的 Rust 代码并把它转换为 `Tokens` 。 `quote!` 让我们可以使用一些炫酷的模板机制；简单的使用 `#name` ， `quote!` 就会把它替换为叫做 `name` 的变量。你甚至可以类似常规宏那样进行一些重复。请查看这些文档，这里有一些好的介绍。

应该就这些了。噢，对了，我们需要在 `hello-world-derive` crate 的 `cargo.toml` 中添加 `syn` 和 `quote` 的依赖。

```
[dependencies]
syn = "0.10.5"
quote = "0.3.10"
```

这样就 OK 了。尝试编译 `hello-world` 。

```
error: the `#[proc_macro_derive]` attribute is only usable with
crates of the `proc-macro` crate type
 --> hello-world-derive/src/lib.rs:8:3
  |
8 | #[proc_macro_derive(HelloWorld)]
  |   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

好吧，看来我们需要把 `hello-world-derive` crate 声明为 `proc-macro` 类型。怎么做呢？像这样：

```
[lib]
proc-macro = true
```

现在好了，编译 `hello-world` 。现在执行 `cargo run` 将会输出：

```
Hello, World! My name is FrenchToast
Hello, World! My name is Waffles
```

我们成功了！

自定义 attribute（Custom Attributes）

在一些情况下允许用户进行一些配置是合理的。例如，用户可能想要重载 `hello_world()` 方法打印出的名字的值。

这可以通过自定义 attribute 来实现：

```
#[derive(HelloWorld)]
#[HelloWorldName = "the best Pancakes"]
struct Pancakes;

fn main() {
    Pancakes::hello_world();
}
```

但是如果我们尝试编译它，编译器会返回一个错误：

```
error: The attribute `HelloWorldName` is currently unknown to th
e compiler and may have meaning added to it in the future (see i
ssue #29642)
```

编译器需要知道我们处理了这个 attribute 才能不返回错误。这可以通过在 `hello-world-derive` crate 中对 `proc_macro_derive` attribute 增加 `attributes` 来实现：

```
#[proc_macro_derive(HelloWorld, attributes(HelloWorldName))]
pub fn hello_world(input: TokenStream) -> TokenStream
```

可以一同样的方式指定多个 attribute。

引发错误

让我们假设我们并不希望在我们的自定义导出方法中接受枚举作为输入。

这个条件可以通过 `syn` 轻松的进行检查。不过我们如何告诉用户，我们并不接受枚举呢？在过程宏中报告错误的传统做法是 panic：

```
fn impl_hello_world(ast: &syn::MacroInput) -> quote::Tokens {
    let name = &ast.ident;
    // Check if derive(HelloWorld) was specified for a struct
    if let syn::Body::Struct(_) = ast.body {
        // Yes, this is a struct
        quote! {
            impl HelloWorld for #name {
                fn hello_world() {
                    println!("Hello, World! My name is {}", stringify!(#name));
                }
            }
        }
    } else {
        //Nope. This is an Enum. We cannot handle these!
       panic!("#[derive(HelloWorld)] is only defined for structs, not for enums!");
    }
}
```

如果用户尝试从一个枚举导出 `HelloWorld` ，他们会收到如下希望有帮助的错误信息：

```
error: custom derive attribute panicked
  --> src/main.rs
   |
   | #[derive(HelloWorld)]
   |          ^^^^^^^^^^
   |
   = help: message: #[derive(HelloWorld)] is only defined for structs, not for enums!
```

Rust开发版

> nightly-rust.md
> commit eb1c7161dd79b55e022cd0c661f9018d406b3fe4

Rust 提供了三种发行渠道：开发版（每日构建），beta 版和稳定版。不稳定功能只在 Rust 开发版中可用。对于这个进程的更多细节，参见可交付产品的稳定性。

要安装 Rust 开发版，你可以使用 `rustup.sh` ：

```
$ curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --ch
annel=nightly
```

如果你担心使用 `curl | sh` 的潜在不安全性，请继续阅读并查看我们下面的免责声明。你也可以进行两步安装，以便于检查我们的安装脚本：

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh --channel=nightly
```

如果你用 Windows，请直接下载32位安装包或者64位安装包然后运行即可。

卸载

如果你决定不再需要 Rust 了，我们会有点难过，但没关系。不是每种编程语言都适合所有人。运行下面的卸载脚本即可：

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

如果你使用 Windows 安装包进行安装的话，重新运行 `.msi` 文件，它会提供卸载选项。

一些同学确实有理由对我们让他们运行 `curl | sudo sh` 感到反感。从根本上说，当你运行上面的脚本时，代表你相信是一些好人在维护 Rust，他们不会黑了你的电脑做坏事。对此保持警觉是很好的天性。如果你是这些人之一，请检阅以下文档：从源码编译Rust或者官方二进制文件下载。

当然，我们还应该提到官方支持的平台：

- Windows（7+）
- Linux（2.6.18 或更高版本，各种发行版），x86 和 x86-64
- OSX 10.7（Lion）或更高版本，x86 和 x86-64

Rust 在以上平台进行了广泛的测试，当然其他一些平台也有，比如 Android。不过在上述平台工作起来最顺畅，因为它们进行了最多的测试。

最后说说 Windows。Rust 将 Windows 作为第一级平台来发布，不过说实话，WIndows 的集成体验并没有 Linux/OS X 那么好。我们正在改进！如果有不能工作的情况，就是出了 bug。一旦发生了请告知我们。任何一次提交都会在 Windows 下进行测试，和其他平台无异。

如果你已安装 Rust，你可以打开一个 Shell，然后输入：

```
$ rustc --version
```

你应该看到版本号，提交的 hash 值，提交时间和构建时间：

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

如果你做到了，那么 Rust 已经成功安装！此处应有掌声！

如果你遇到什么错误，有几个你可以获取帮助的地方。最简单的是通过 Mibbit 访问 Rust IRC 频道 irc.mozilla.org。点击上面的链接，你就可以与其他 Rustacean（我们这些人自称的绰号）聊天了，我们会帮助你。其他给力的资源包括用户论坛和 Stack Overflow。

编译器插件

> compiler-plugins.md
> commit 28548db57d0acbc00ee80b43816953dbe31d53ba

介绍

`rustc` 可以加载编译器插件，它是由用户提供的库用来扩充编译器的行为，例如新的语法扩展，lint检查等。

一个插件是带有设计好的用来在 `rustc` 中注册扩展的注册（*registrar*）函数的一个动态库包装箱。其它包装箱可以使用 `#![plugin(...)]` 属性来装载这个扩展。查看rustc::plugin文档来获取更多关于定义和装载插件的机制。

如果属性存在的话， `#![plugin(foo(... args ...))]` 传递的参数并不由 `rustc` 自身解释。它们被传递给插件的 `Registry` args方法。

在绝大多数情况中，一个插件应该只通过 `#![plugin]` 而不通过 `extern crate` 来使用。链接一个插件会将 `libsyntax` 和 `librustc` 加入到你的包装箱的依赖中。基本上你不会希望如此除非你在构建另一个插件。 `plugin_as_library` lint会检查这些原则。

通常的做法是将插件放到它们自己的包装箱中，与任何那些会被库的调用者使用的 `macro_rules!` 宏或 Rust 代码分开。

语法扩展

插件可以有多种方法来扩展 Rust 的语法。一种语法扩展是宏过程。它们与普通宏的调用方法一样，不过扩展是通过执行任意Rust代码在编译时操作语法树进行的。

让我们写一个实现了罗马数字的插件roman_numerals.rs。

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
```

```
extern crate rustc;
extern crate rustc_plugin;

use syntax::parse::token;
use syntax::tokenstream::TokenTree;
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEage
r};
use syntax::ext::build::AstBuilder;  // A trait for expr_usize.
use syntax::ext::quote::rt::Span;
use rustc_plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult + 'static> {

    static NUMERALS: &'static [(&'static str, usize)] = &[
        ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
        ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
        ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
        ("I",    1)];

    if args.len() != 1 {
        cx.span_err(
            sp,
            &format!("argument should be a single identifier, bu
t got {} arguments", args.len()));
        return DummyResult::any(sp);
    }

    let text = match args[0] {
        TokenTree::Token(_, token::Ident(s)) => s.to_string(),
        _ => {
            cx.span_err(sp, "argument should be a single identif
ier");
            return DummyResult::any(sp);
        }
    };

    let mut text = &*text;
    let mut total = 0;
    while !text.is_empty() {
```

```
        match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(
rn)) {
            Some(&(rn, val)) => {
                total += val;
                text = &text[rn.len()..];
            }
            None => {
                cx.span_err(sp, "invalid Roman numeral");
                return DummyResult::any(sp);
            }
        }
    }

    MacEager::expr(cx.expr_usize(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("rn", expand_rn);
}
```

我们可以像其它宏那样使用 `rn!()` ：

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

与一个简单的 `fn(&str) -> u32` 函数相比的优势有：

- （任意复杂程度的）转换都发生在编译时
- 输入验证也在编译时进行
- 可以扩展并允许在模式中使用，它可以有效的为任何数据类型定义新语法。

除了宏过程，你可以定义新的类derive属性和其它类型的扩展。查看Registry::register_syntax_extension和SyntaxExtension enum。对于更复杂的宏例子，查看regex_macros。

提示与技巧

这里提供一些宏调试的提示。

你可以使用syntax::parse来将记号树转换为像表达式这样的更高级的语法元素：

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult+'static> {

    let mut parser = cx.new_parser_from_tts(args);

    let expr: P<Expr> = parser.parse_expr();
```

看完libsyntax解析器代码会给你一个解析基础设施如何工作的感觉。

保留你解析所有的Span，以便更好的报告错误。你可以用Spanned包围你的自定数据结构。

调用ExtCtxt::span_fatal将会立即终止编译。相反最好调用ExtCtxt::span_err并返回DummyResult，这样编译器可以继续并找到更多错误。

为了打印用于调试的语法段，你可以同时使用span_note和syntax::print::pprust::*_to_string。

上面的例子使用AstBuilder::expr_usize产生了一个普通整数。作为一个 `AstBuilder` 特性的额外选择， `libsyntax` 提供了一个准引用宏的集合。它们并没有文档并且非常边缘化。然而，这些将会是实现一个作为一个普通插件库的改进准引用的好的出发点。

Lint 插件

插件可以扩展Rust Lint基础设施来添加额外的代码风格，安全检查等。你可以查看src/test/auxiliary/lint_plugin_test.rs来了解一个完整的例子，我们在这里重现它的核心部分：

```
#![feature(plugin_registrar)]
#![feature(box_syntax, rustc_private)]
```

```
extern crate syntax;

// Load rustc as a plugin to get macros
#[macro_use]
extern crate rustc;
extern crate rustc_plugin;

use rustc::lint::{EarlyContext, LintContext, LintPass, EarlyLint
Pass,
                  EarlyLintPassObject, LintArray};
use rustc_plugin::Registry;
use syntax::ast;

declare_lint!(TEST_LINT, Warn, "Warn about items named 'lintme'"
);

struct Pass;

impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_LINT)
    }
}

impl EarlyLintPass for Pass {
    fn check_item(&mut self, cx: &EarlyContext, it: &ast::Item)
{
        if it.ident.name.as_str() == "lintme" {
            cx.span_lint(TEST_LINT, it.span, "item is named 'lin
tme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_early_lint_pass(box Pass as EarlyLintPassObject
);
}
```

那么像这样的代码：

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

将产生一个编译警告：

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

Lint插件的组件有：

- 一个或多个 `declare_lint!` 调用，它定义了Lint结构
- 一个用来存放lint检查所需的所有状态（在我们的例子中，没有）
- 一个定义了如何检查每个语法元素的LintPass实现。一个单独的 `LintPass` 可能会对多个不同的 `Lint` 调用 `span_lint` ，不过它们都需要用 `get_lints` 方法进行注册。

Lint过程是语法遍历，不过它们运行在编译的晚期，这时类型信息时可用的。 `rustc` 的内建lint与lint插件使用相同的基础构架，并提供了如何访问类型信息的例子。

由插件定义的语法通常通过属性和插件标识控制，例如， `[#[allow(test_lint)]]` ， `-A test-lint` 。这些标识符来自于 `declare_lint!` 的第一个参数，经过合适的大小写和标点转换。

你可以运行 `rustc -W help foo.rs` 来见检查lint列表是否为 `rustc` 所知，包括由 `foo.rs` 加载的插件。

内联汇编

> inline-assembly.md
> commit 28548db57d0acbc00ee80b43816953dbe31d53ba

为了极端底层操作和性能要求，你可能希望直接控制 CPU。Rust 通过 `asm!` 宏来支持使用内联汇编。

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

任何 `asm` 的使用需要功能通道（需要在包装箱上加上 `#![feature(asm)]` 来允许使用）并且当然也需要写在 `unsafe` 块中

> 注意：这里的例子使用了 x86/x86-64 汇编，不过所有平台都受支持。

汇编模板

`assembly template` 是唯一需要的参数并且必须是原始字符串（就是 `""` ）

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
    unsafe {
        asm!("NOP");
    }
}

// Other platforms:
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
    // ...
    foo();
    // ...
}
```

（`feature(asm)` 和 `#[cfg]` 从现在开始将被忽略。）

输出操作数，输入操作数，覆盖和选项都是可选的，然而如果你要省略它们的话，你必选加上正确数量的 `:` ：

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax"
    :
    :
    : "eax"
   );
# } }
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn main() {}
```

有空格在中间也没关系：

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax" ::: "eax");
# } }
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn main() {}
```

操作数

输入和输出操作数都有相同的格式：`: "constraints1"(expr1), "constraints2"(expr2), ..."`。输出操作数表达式必须是可变的左值，或还未赋值的：

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
             );
    }
    c
}
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn add(a: i32, b: i32) -> i32 { a + b }

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

如果你想在这里使用真正的操作数，然而，要求你在你想使用的寄存器上套上大括号 `{}`，并且要求你指明操作数的大小。这在非常底层的编程中是很有用的，这时你使用哪个寄存器是很重要的：

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# unsafe fn read_byte_in(port: u16) -> u8 {
let result: u8;
asm!("in %dx, %al" : "={al}"(result) : "{dx}"(port));
result
# }
```

覆盖（Clobbers）

一些指令修改的寄存器可能保存有不同的值，所以我们使用覆盖列表来告诉编译器不要假设任何装载在这些寄存器的值是有效的。

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
// Put the value 0x200 in eax:
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ :
"eax");
# } }
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn main() {}
```

输入和输出寄存器并不需要列出因为这些信息已经通过给出的限制沟通过了。因此，任何其它的被使用的寄存器应该隐式或显式的被列出。

如果汇编修改了代码状态寄存器 `cc` 则需要在覆盖中被列出，如果汇编修改了内存，`memory` 也应被指定。

选项（Options）

最后一部分，`options` 是 Rust 特有的。格式是逗号分隔的基本字符串（也就是说，`:"foo", "bar", "baz"`）。它被用来指定关于内联汇编的额外信息：

目前有效的选项有：

1. *volatile* - 相当于 gcc/clang 中的 `__asm__ __volatile__ (...)`

2. *alignstack* - 特定的指令需要栈按特定方式对齐（比如，SSE）并且指定这个告诉编译器插入通常的栈对齐代码
3. *intel* - 使用 intel 语法而不是默认的 AT&T 语法

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() {
let result: i32;
unsafe {
   asm!("mov eax, 2" : "={eax}"(result) : : : "intel")
}
println!("eax is currently {}", result);
# }
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn main() {}
```

更多信息

目前 `asm!` 的实现是一个LLVM内联汇编表达式的直接绑定，所以请确保充分的阅读他们的文档来获取关于覆盖，限制等概念的更多信息。

不使用标准库

> no-stdlib.md
> commit 893f42a83466cf02b6fd6d3c82d5419cdad47474

Rust 的标准库提供了很多有用的功能，不过它假设它的 host 系统的多种功能的支持：线程，网络，堆分配和其他功能。有些系统并没有这些功能，不过，Rust也能在这些系统上工作。为此，我们可以通过一个属性来告诉 Rust 我们不想使用标准库： `#![no_std]` 。

> 注意：这个功能技术上是稳定的，不过有些附加条件。其一，你可以构建一个稳定的 `#![no_std]` 库，但二进制文件不行。关于没有标准库的库文件的细节，查看关于 `#![no_std]` 的章节。

很显然你并不一定需要标准库：可以使用 `#[no_std` 来构建一个可执行程序。

使用 libc

为了构建一个 `#[no_std]` 可执行程序，我们需要 libc 作为依赖。可以在 `Cargo.toml` 文件中指定：

```
[dependencies]
libc = { version = "0.2.14", default-features = false }
```

注意默认功能被禁用了。这是关键的一步————**libc** 的默认功能引用了标准库所以必须被禁用。

不用标准库编写可执行程序

有两种可能的控制入口点的方法： `#[start]` 属性，或者用你自己的代码 override C `main` 函数的默认 shim。

被标记为 `#[start]` 的函数传递的参数格式与 C 一致：

```
#![feature(lang_items, core_intrinsics)]
#![feature(start)]
#![no_std]
use core::intrinsics;

// Pull in the system libc library for what crt0.o likely requir
es.
extern crate libc;

// Entry point for this program.
#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    0
}

// These functions are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "eh_personality"]
#[no_mangle]
pub extern fn rust_eh_personality() {
}

// This function may be needed based on the compilation target.
#[lang = "eh_unwind_resume"]
#[no_mangle]
pub extern fn rust_eh_unwind_resume() {
}

#[lang = "panic_fmt"]
#[no_mangle]
pub extern fn rust_begin_panic(_msg: core::fmt::Arguments,
                               _file: &'static str,
                               _line: u32) -> ! {
    unsafe { intrinsics::abort() }
}
```

要 override 编译器插入的 `main` shim，你必须使用 `#![no_main]` 禁用它并通过正确的 ABI 和正确的名字来创建合适的函数，这也需要需要覆盖编译器的命名改编：

```
#![feature(lang_items, core_intrinsics)]
#![feature(start)]
#![no_std]
#![no_main]
use core::intrinsics;

// Pull in the system libc library for what crt0.o likely requir
es.
extern crate libc;

// Entry point for this program.
#[no_mangle] // ensure that this symbol is called `main` in the
output
pub extern fn main(_argc: i32, _argv: *const *const u8) -> i32 {
    0
}

// These functions are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "eh_personality"]
#[no_mangle]
pub extern fn rust_eh_personality() {
}

// This function may be needed based on the compilation target.
#[lang = "eh_unwind_resume"]
#[no_mangle]
pub extern fn rust_eh_unwind_resume() {
}

#[lang = "panic_fmt"]
#[no_mangle]
pub extern fn rust_begin_panic(_msg: core::fmt::Arguments,
                               _file: &'static str,
                               _line: u32) -> ! {
    unsafe { intrinsics::abort() }
}
```

关于 language items 的更多细节

目前编译器对能够被可执行文件调用的符号做了一些假设。正常情况下，这些函数是由标准库提供的，不过没有它你就必须定义你自己的了。这些符号被称为“language items”，并且他们每个都有一个内部的名称，和一个必须符合签名的实现。

这些函数中的第一个，`eh_personality`，被编译器的错误机制使用。它通常映射到 GCC 的特性函数上（查看libstd实现来获取更多信息），不过对于不会触发恐慌的包装箱可以确定这个函数不会被调用。language item 的名称是`eh_personality`。

第二个函数，`rust_begin_panic`，也被作为编译器的错误机制使用。当发生 panic 时，它控制显示在屏幕上的信息。虽然 language item 的名称是`panic_fmt`，但是符号的名称是`rust_begin_panic`。

第三个函数，`rust_eh_unwind_resume`，在target 选项中的`custom_unwind_resume` flag 被设置时也是必需的。它允许自定义在 landing pads 的最后的 resuming unwind 过程。language item 的名字是`eh_unwind_resume`。

固有功能

> intrinsics.md
> commit 024aa9a345e92aa1926517c4d9b16bd83e74c10d
>
> 注意：固有功能将会永远是一个不稳定的接口，推荐使用稳定的 libcore 接口而不是直接使用编译器自带的功能。

可以像 FFI 函数那样导入它们，使用特殊的 `rust-intrinsic` ABI。例如，如果在一个独立的上下文，但是想要能在类型间 `transmute` ，并想进行高效的指针计算，你可以声明函数：

```
#![feature(intrinsics)]
# fn main() {}

extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

跟其它 FFI 函数一样，它们总是 `unsafe` 的。

语言项（Lang items）

> lang-items.md
> commit 893f42a83466cf02b6fd6d3c82d5419cdad47474
>
> 注意：语言项通常由 Rust 发行版的 crate 提供，并且它自身有一个不稳定的接口。建议使用官方发布的 crate 而不是定义自己的版本。

`rustc` 编译器有一些可插入的操作，也就是说，功能不是硬编码进语言的，而是在库中实现的，通过一个特殊的标记告诉编译器它存在。这个标记是 `#[lang="..."]` 属性并且有不同的值 `...`,也就是不同的“语言项”。

例如， `Box` 指针需要两个语言项，一个用于分配，一个用于释放。下面是一个独立的程序使用 `Box` 语法糖进行动态分配，通过 `malloc` 和 `free` ：

```
#![feature(lang_items, box_syntax, start, libc, core_intrinsics)
]
#![no_std]
use core::intrinsics;

extern crate libc;

#[lang = "owned_box"]
pub struct Box<T>(*mut T);

#[lang = "exchange_malloc"]
unsafe fn allocate(size: usize, _align: usize) -> *mut u8 {
    let p = libc::malloc(size as libc::size_t) as *mut u8;

    // Check if `malloc` failed:
    if p as usize == 0 {
        intrinsics::abort();
    }

    p
}

#[lang = "exchange_free"]
```

```
unsafe fn deallocate(ptr: *mut u8, _size: usize, _align: usize)
{
    libc::free(ptr as *mut libc::c_void)
}

#[lang = "box_free"]
unsafe fn box_free<T: ?Sized>(ptr: *mut T) {
    deallocate(ptr as *mut u8, ::core::mem::size_of_val(&*ptr),
::core::mem::align_of_val(&*ptr));
}

#[start]
fn main(argc: isize, argv: *const *const u8) -> isize {
    let x = box 1;

    0
}

#[lang = "eh_personality"] extern fn rust_eh_personality() {}
#[lang = "panic_fmt"] extern fn rust_begin_panic() -> ! { unsafe
 { intrinsics::abort() }
# #[lang = "eh_unwind_resume"] extern fn rust_eh_unwind_resume()
 {}
# #[no_mangle] pub extern fn rust_eh_register_frames () {}
# #[no_mangle] pub extern fn rust_eh_unregister_frames () {}
```

注意 `abort` 的使用： `exchange_malloc` 语言项假设返回一个有效的指针，所以需要在内部进行检查。

其它语言项提供的功能包括：

- 通过特性重载运算符： `==` ， `<` ，解引用（ `*` ）和 `+` 等运算符对应的特性都有语言项标记；上面4个分别为 `eq` ， `ord` ， `deref` 和 `add`
- 栈展开和一般故障： `eh_personality` ， `fail` 和 `fail_bounds_checks` 语言项
- `std::marker` 中用来标明不同类型的特性： `send` ， `sync` 和 `copy` 。
- `std::marker` 中的标记类型和变化指示器： `covariant_type` 和 `contravariant_lifetime` 等

语言项由编译器延时加载；例如，如果你从未用过 `Box` 则就没有必要定义 `exchange_malloc` 和 `exchange_free` 的函数。 `rustc` 在一个项被需要而无法在当前包装箱或任何依赖中找到时生成一个错误。

链接进阶

> advanced-linking.md
> commit 226bcdf7d1e774f5967f92b0bd0bf237179f95c9

Rust 的常用链接形式在本书的之前部分已经介绍过了，不过支持多种其他语言可用的可能的链接对 Rust 获取与原生库的无缝交互是很重要的。

链接参数（Link args）

这里还有一个方法来告诉 rustc 如何自定义链接，这就是通过 `link_args` 属性。这个属性作用于 `extern` 块并指定当产生构件时需要传递给连接器的原始标记。一个用例将是：

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
# fn main() {}
```

注意现在这个功能隐藏在 `feature(link_args)` gate 之后因为它并不是一个被认可的执行链接的方法。目前 rustc 从 shell 调用系统的连接器（大多数系统是 `gcc`，MSVC 是 `link.exe`），所以使用额外的命令行参数是可行的，不过这并一定永远可行。将来 rustc 可能使用 LLVM 直接链接原生库这样一来 `link_args` 就毫无意义了。你可以向 `rustc` 传递 `-C link-args` 参数来获得和 `link_args` 属性同样的效果。

强烈建议你不要使用这个属性，而是使用一个更正式的 `[link(...)]` 属性作用于 `extern` 块。

静态链接

静态链接代表创建包含所有所需库的输出的过程，这样你在任何系统上使用你编译的项目时就不需要安装相应的库了。纯 Rust 的依赖默认都是静态链接的这样你可以使用你创建的二进制和库而不需要安装 Rust。相反，原生库（例

如，`libc` 和 `libm`）通常是动态链接的，不过也可以修改为静态链接。

链接是一个非常依赖平台的问题--在一些平台上，静态链接可能根本就是不可能的！这个部分假设你对你选择的平台的链接一些基础的认识。

Linux

在 Linux 上 Rust 程默认会链接系统的 `libc` 以及一些其他的库。让我们看看一个使用 GCC 和 `glibc` 的 64 位 Linux（目前为止 Linux 上最常见的 `libc`）的例子：

```
$ mkdir musldist
$ PREFIX=$(pwd)/musldist
$
$ # Build musl
$ curl -O http://www.musl-libc.org/releases/musl-1.1.10.tar.gz
$ tar xf musl-1.1.10.tar.gz
$ cd musl-1.1.10/
musl-1.1.10 $ ./configure --disable-shared --prefix=$PREFIX
musl-1.1.10 $ make
musl-1.1.10 $ make install
musl-1.1.10 $ cd ..
$ du -h musldist/lib/libc.a
2.2M    musldist/lib/libc.a
$
$ # Build libunwind.a
$ curl -O http://llvm.org/releases/3.7.0/llvm-3.7.0.src.tar.xz
$ tar xf llvm-3.7.0.src.tar.xz
$ cd llvm-3.7.0.src/projects/
llvm-3.7.0.src/projects $ curl http://llvm.org/releases/3.7.0/li
bunwind-3.7.0.src.tar.xz | tar xJf -
llvm-3.7.0.src/projects $ mv libunwind-3.7.0.src libunwind
llvm-3.7.0.src/projects $ mkdir libunwind/build
llvm-3.7.0.src/projects $ cd libunwind/build
llvm-3.7.0.src/projects/libunwind/build $ cmake -DLLVM_PATH=../.
./.. -DLIBUNWIND_ENABLE_SHARED=0 ..
llvm-3.7.0.src/projects/libunwind/build $ make
llvm-3.7.0.src/projects/libunwind/build $ cp lib/libunwind.a $PR
EFIX/lib/
```

```
llvm-3.7.0.src/projects/libunwind/build $ cd ../../../../
$ du -h musldist/lib/libunwind.a
164K    musldist/lib/libunwind.a
$
$ # Build musl-enabled rust
$ git clone https://github.com/rust-lang/rust.git muslrust
$ cd muslrust
muslrust $ ./configure --target=x86_64-unknown-linux-musl --musl
-root=$PREFIX --prefix=$PREFIX
muslrust $ make
muslrust $ make install
muslrust $ cd ..
$ du -h musldist/bin/rustc
12K     musldist/bin/rustc
```

现在你有了一个启用了 `musl` 的Rust！因为我们用了一个自定义的目录，当我们尝试并运行它的时候我们需要确保我们的系统能够找到二进制文件和正确的库：

```
$ export PATH=$PREFIX/bin:$PATH
$ export LD_LIBRARY_PATH=$PREFIX/lib:$LD_LIBRARY_PATH
```

让我们试一下！

```
$ echo 'fn main() { println!("hi!"); panic!("failed"); }' > exam
ple.rs
$ rustc --target=x86_64-unknown-linux-musl example.rs
$ ldd example
        not a dynamic executable
$ ./example
hi!
thread 'main' panicked at 'failed', example.rs:1
```

成功了！这个二进制文件可以被拷贝到几乎所有拥有相同构架的 Linux 机器上无故障的运行。

`cargo build` 也允许 `--target` 选项所以你也能用它来正常的构建你的 crate。然而，你可能需要先链接你的原生库到 `musl` ，在你可以链接到它之前。

基准测试

> benchmark-tests.md
> commit 28548db57d0acbc00ee80b43816953dbe31d53ba

Rust 也支持基准测试，它可以测试代码的性能。让我们把 `src/lib.rs` 修改成这样（省略注释）：

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

注意 `test` 功能 gate，它启用了这个不稳定功能。

我们导入了 `test` crate，它包含了对基准测试的支持。我们也定义了一个新函数，带有 `bench` 属性。与一般的不带参数的测试不同，基准测试有一个 `&mut Bencher` 参数。 `Bencher` 提供了一个 `iter` 方法，它接收一个闭包。这个闭包包

含我们想要测试的代码。

我们可以用 `cargo bench` 来运行基准测试：

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

我们的非基准测试将被忽略。你也许会发现 `cargo bench` 比 `cargo test` 花费的时间更长。这是因为Rust会多次运行我们的基准测试，然后取得平均值。因为我们的函数只做了非常少的操作，我们耗费了 `1 ns/iter (+/- 0)` ，不过运行时间更长的测试就会有出现偏差。

编写基准测试的建议：

- 把初始代码放于 `iter` 循环之外，只把你想要测试的部分放入它
- 确保每次循环都做了"同样的事情"，不要累加或者改变状态
- 确保外边的函数也是幂等的（idempotent），基准测试runner可能会多次运行它
- 确保 `iter` 循环内简短而快速，这样基准测试会运行的很快同时校准器可以在合适的分辨率上调整运转周期
- 确保 `iter` 循环执行简单的工作，这样可以帮助我们准确的定位性能优化（或不足）

Gocha：优化

写基准测试有另一些比较微妙的地方：开启了优化编译的基准测试可能被优化器戏剧性的修改导致它不再是我们期望的基准测试了。举例来说，编译器可能认为一些计算并无外部影响并且整个移除它们。

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new);
    });
}
```

得到如下结果：

```
running 1 test
test bench_xor_1000_ints ... bench:         0 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

基准测试运行器提供两种方法来避免这个问题：要么传递给 `iter` 的闭包可以返回一个随机的值这样强制优化器认为结果有用并确保它不会移除整个计算部分。这可以通过修改上面例子中的 `b.iter` 调用：

```
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} }
let b = X;
b.iter(|| {
    // Note lack of `;` (could also use an explicit `return`).
    (0..1000).fold(0, |old, new| old ^ new)
});
```

要么，另一个选择是调用通用的 `test::black_box` 函数，它会传递给优化器一个不透明的“黑盒”这样强制它考虑任何它接收到的参数。

```
#![feature(test)]

extern crate test;

# fn main() {
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} }
let b = X;
b.iter(|| {
    let n = test::black_box(1000);

    (0..n).fold(0, |a, b| a ^ b)
})
# }
```

上述两种方法均未读取或修改值，并且对于较小的值来说非常廉价。对于较大的值可以通过间接传递来减小额外开销（例如： `black_box(&huge_struct)` ）。

执行上面任何一种修改可以获得如下基准测试结果：

```
running 1 test
test bench_xor_1000_ints ... bench:       131 ns/iter (+/- 3)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

然而，即使使用了上述方法优化器还是可能在不合适的情况下修改测试用例。

装箱语法和模式

> box-syntax-and-patterns.md
>
> commit 28548db57d0acbc00ee80b43816953dbe31d53ba

目前唯一稳定的创建 `Box` 的方法是通过 `Box::new` 方法。并且不可能在一个模式匹配中稳定的析构一个 `Box` 。不稳定的 `box` 关键字可以用来创建和析构 `Box` 。下面是一个用例：

```
#![feature(box_syntax, box_patterns)]

fn main() {
    let b = Some(box 5);
    match b {
        Some(box n) if n < 0 => {
            println!("Box contains negative number {}", n);
        },
        Some(box n) if n >= 0 => {
            println!("Box contains non-negative number {}", n);
        },
        None => {
            println!("No box");
        },
        _ => unreachable!()
    }
}
```

注意这些功能目前隐藏在 `box_syntax` （装箱创建）和 `box_patterns` （析构和模式匹配）gate 之后因为它的语法在未来可能会改变。

返回指针

在很多有指针的语言中，你的函数可以返回一个指针来避免拷贝大的数据结构。例如：

```
struct BigStruct {
    one: i32,
    two: i32,
    // Etc.
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> Box<BigStruct> {
    Box::new(*x)
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y = foo(x);
}
```

要点是通过传递一个装箱，你只需拷贝了一个指针，而不是那构成了 `BigStruct` 的一百个 `int` 值。

上面是 Rust 中的一个反模式。相反，这样写：

```
#![feature(box_syntax)]

struct BigStruct {
    one: i32,
    two: i32,
    // Etc.
    one_hundred: i32,
}

fn foo(x: Box<BigStruct>) -> BigStruct {
    *x
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y: Box<BigStruct> = box foo(x);
}
```

这在不牺牲性能的前提下获得了灵活性。

你可能会认为这会给我们带来很差的性能：返回一个值然后马上把它装箱？难道这在哪里不都是最糟的吗？Rust 显得更聪明。这里并没有拷贝。`main` 为装箱分配了足够的空间，向 `foo` 传递一个指向他内存的 `x`，然后 `foo` 直接向 `Box<T>` 中写入数据。

因为这很重要所以要说两遍：返回指针会阻止编译器优化你的代码。允许调用函数选择它们需要如何使用你的输出。

切片模式

> slice-patterns.md
> commit 5cf4139d21073731fa7f7226d941349dbacc16d6

如果你想在一个切片或数组上匹配，你可以通过 `slice_patterns` 功能使用 `&` ：

```
#![feature(slice_patterns)]

fn main() {
    let v = vec!["match_this", "1"];

    match &v[..] {
        &["match_this", second] => println!("The second element
is {}", second),
        _ => {},
    }
}
```

`advanced_slice_patterns` gate 让你使用 `..` 表明在一个切片的模式匹配中任意数量的元素。这个通配符对一个给定的数组只能只用一次。如果在 `..` 之前有一个标识符，结果会被绑定到那个名字上。例如：

```
#![feature(advanced_slice_patterns, slice_patterns)]

fn is_symmetric(list: &[u32]) -> bool {
    match list {
        &[] | &[_] => true,
        &[x, ref inside.., y] if x == y => is_symmetric(inside),
        _ => false
    }
}

fn main() {
    let sym = &[0, 1, 4, 2, 4, 1, 0];
    assert!(is_symmetric(sym));

    let not_sym = &[0, 1, 7, 2, 4, 1, 0];
    assert!(!is_symmetric(not_sym));
}
```

关联常量

> associated-constants.md
> commit 024aa9a345e92aa1926517c4d9b16bd83e74c10d

通过 `associated_consts` 功能，你像这样可以定义常量：

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
    const ID: i32 = 1;
}

fn main() {
    assert_eq!(1, i32::ID);
}
```

任何 `Foo` 的定义都必须定义 `ID` ，不定义的话：

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
}
```

会给出

```
error: not all trait items implemented, missing: `ID` [E0046]
     impl Foo for i32 {
     }
```

也可以实现一个默认值：

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32 = 1;
}

impl Foo for i32 {
}

impl Foo for i64 {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);
    assert_eq!(5, i64::ID);
}
```

如你所见，当实现 `Foo` 时，你可以不实现它（关联常量），当作为 `i32` 时。接着它将会使用默认值。不过，作为 `i64` 时，我们可以增加我们自己的定义。

关联常量并不一定要关联在一个 trait 上。一个 `struct` 的 `impl` 块或 `enum` 也行：

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
    const FOO: u32 = 3;
}
```

自定义内存分配器

> custom-allocators.md
> commit 28548db57d0acbc00ee80b43816953dbe31d53ba

分配内存并不总是最简单的事情，同时通常 Rust 默认会负责它，不过经常自定义内存分配会变得必要。编译器和标准库目前允许在编译时切换目前默认使用的全局分配器。设计目前称作RFC 1183不过这里我们会教你如何获取你自己的分配器并运行起来。

默认分配器

编译器目前自带两个默认分配器：`alloc_system` 和 `alloc_jemalloc` （然而一些目标平台并没有 jemalloc）。这些分配器是正常的 Rust crate 并包含分配和释放内存的 routine 的实现。标准库并不假设使用任何一个编译，而且编译器会在编译时根据被产生的输出类型决定使用哪个分配器。

编译器产生的二进制文件默认会使用 `alloc_jemalloc` （如果可用的话）。在这种情况下编译器“控制了一切”，从它超过了最终链接的权利的角度来看。大体上这意味着分配器选择可以被交给编译器。

动态和静态库，然而，默认使用 `alloc_system` 。这里 Rust 通常是其他程序的“客人”或者处于并没有权决定应使用的分配器的世界。为此它求助于标准 API（例如， `malloc` 和 `free` ）来获取和释放内存。

切换分配器

虽然编译器默认的选择大部分情况工作良好，也经常需要定制特定的方面。覆盖编译器关于使用哪个分配器的选择可以简单的通过链接到期望的分配器实现：

```
#![feature(alloc_system)]

extern crate alloc_system;

fn main() {
    let a = Box::new(4); // Allocates from the system allocator.
    println!("{}", a);
}
```

在这个例子中生成的二进制文件并不会默认链接到 jemalloc 而是使用了系统分配器。同理生成一个默认使用 jemalloc 的动态库可以写成：

```
#![feature(alloc_jemalloc)]
#![crate_type = "dylib"]

extern crate alloc_jemalloc;

pub fn foo() {
    let a = Box::new(4); // Allocates from jemalloc.
    println!("{}", a);
}
# fn main() {}
```

编写一个自定义分配器

有时甚至 jemalloc 与系统分配器之间的选择都是不够的并需要一个新的自定义的分配器。这种情况你要编写你自己实现了分配器 API（例如与 `alloc_system` 和 `alloc_jemallo` 相同）的 crate。作为一个例子，让我们看看一个简单的和声明化的 `alloc_system` 版本：

```
# // Only needed for rustdoc --test down below.
# #![feature(lang_items)]
// The compiler needs to be instructed that this crate is an allocator in order
// to realize that when this is linked in another allocator like jemalloc should
// not be linked in.
```

```
#![feature(allocator)]
#![allocator]

// Allocators are not allowed to depend on the standard library
which in turn
// requires an allocator in order to avoid circular dependencies
. This crate,
// however, can use all of libcore.
#![no_std]

// Let's give a unique name to our custom allocator:
#![crate_name = "my_allocator"]
#![crate_type = "rlib"]

// Our system allocator will use the in-tree libc crate for FFI
bindings. Note
// that currently the external (crates.io) libc cannot be used b
ecause it links
// to the standard library (e.g. `#![no_std]` isn't stable yet),
 so that's why
// this specifically requires the in-tree version.
#![feature(libc)]
extern crate libc;

// Listed below are the five allocation functions currently requ
ired by custom
// allocators. Their signatures and symbol names are not current
ly typechecked
// by the compiler, but this is a future extension and are requi
red to match
// what is found below.
//
// Note that the standard `malloc` and `realloc` functions do no
t provide a way
// to communicate alignment so this implementation would need to
 be improved
// with respect to alignment in that aspect.

#[no_mangle]
pub extern fn __rust_allocate(size: usize, _align: usize) -> *mut
```

```
 u8 {
    unsafe { libc::malloc(size as libc::size_t) as *mut u8 }
}

#[no_mangle]
pub extern fn __rust_deallocate(ptr: *mut u8, _old_size: usize,
_align: usize) {
    unsafe { libc::free(ptr as *mut libc::c_void) }
}

#[no_mangle]
pub extern fn __rust_reallocate(ptr: *mut u8, _old_size: usize,
size: usize,
                                _align: usize) -> *mut u8 {
    unsafe {
        libc::realloc(ptr as *mut libc::c_void, size as libc::si
ze_t) as *mut u8
    }
}

#[no_mangle]
pub extern fn __rust_reallocate_inplace(_ptr: *mut u8, old_size:
 usize,
                                        _size: usize, _align: us
ize) -> usize {
    old_size // This api is not supported by libc.
}

#[no_mangle]
pub extern fn __rust_usable_size(size: usize, _align: usize) ->
usize {
    size
}

# // Only needed to get rustdoc to test this:
# fn main() {}
# #[lang = "panic_fmt"] fn panic_fmt() {}
# #[lang = "eh_personality"] fn eh_personality() {}
# #[lang = "eh_unwind_resume"] extern fn eh_unwind_resume() {}
# #[no_mangle] pub extern fn rust_eh_register_frames () {}
```

```
# #[no_mangle] pub extern fn rust_eh_unregister_frames () {}
```

在我们编译了这个 crate 之后，他可以被如下使用：

```
extern crate my_allocator;

fn main() {
    let a = Box::new(8); // Allocates memory via our custom allo
cator crate.
    println!("{}", a);
}
```

自定义分配器的限制

使用自定义分配器时要满足一些限制，否则可能导致编译器错误：

- 任何一个程序只能链接到一个分配器。二进制、动态库和静态库必须正好链接到一个分配器上，并且，如果没有显式选择，编译器会选择一个。另一方面，rlib 并不需要链接到一个分配器（不过仍然可以）。
- 一个标记为 `#![needs_allocator]` （例如，目前的 `liballoc` ）的分配器使用者和一个 `#[allocator]` crate 不能直接依赖一个需要分配器的 crate（例如，循环引用是不允许的）。这基本上意味着目前分配器必须依赖 libcore。

词汇表

> glossary.md
> commit de288b41b5950c5075236c08c55b1b0ab2b35542

不是每位 Rustacean 都是系统编程或计算机科学背景的，所以我们加上了可能难以理解的词汇解释。

数量（Arity）

Arity代表函数或操作所需的参数数量。

```rust
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

在上面的例子中 `x` 和 `y` 的Arity是 `2` ，`z` 的Arity是 `3` 。

抽象语法树（Abstract Syntax Tree）

当一个编译器编译你程序的时候，它做了很多不同的事。其中之一就是将你程序中的文本转换为一个‘抽象语法树’，或者‘AST’。这个树是你程序结构的表现。例如，`2 + 3` 可以转换为一个树：

```
  +
 / \
2   3
```

而 `2 + (3 * 4)` 看起来像这样：

```
  +
 / \
2   *
   / \
  3   4
```

参数数量（Arity）

Arity 代表一个函数或操作获取的参数的数量。

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

在上面这个例子中 `x` 和 `y` 的arity是 2。 `z` 的arity是 3。

界限（Bounds）

界限是一个类型或trait的限制。例如，如果界限位于函数参数，那么传递给函数的参数类型必须遵守这个限制。

连接符（Combinators）

连接符是一种高级函数，它只作用于函数和其之前定义的连接符，通过其参数来提供一个结果。可以被用来以一种模块化的方式来控制流程。

动态大小类型（DST (Dynamically Sized Type)）

一个没有静态大小或对齐的类型。（更多信息）

表达式（Expression）

在计算机编程中，一个表达式是一个值，常量，变量，运算符和函数的组合，它可以产生一个单一的值。例如，`2 + (3 * 4)` 是一个返回14的表达式。值得注意的是表达式可以产生副作用。例如，一个表达式中的函数可能会执行一些不只是简单的返回一个值的操作。

面向表达式语言（Expression-Oriented Language）

在早期的编程语言中，表达式和语句时两个不同句法范畴：表达式有一个值而语句做一件事。然而，之后的语言模糊了这个区别，允许表达式执行操作而让语句有一个值。在一个面向表达式的语言中，（几乎）所有语句都是一个表达式并因此返回一个值。由此，这些表达式自身也可以是更大表达式的一部分。

语句（Statement）

在计算机编程中，一个语句是一个编程语言能让计算执行操的最小的独立元素。

语法索引

> syntax-index.md
> commit 23a7a7bdb6a6a43cd7efdd9176b1d3f75d9d0e70

关键词（Keywords）

- `as`：原始的类型转换。详见类型转换（`as`）。
- `break`：退出循环。详见循环（提早结束迭代）。
- `const`：常量和常量裸指针。详见`const`和`static`，裸指针。
- `continue`：继续进行下一次迭代。详见循环（提早结束迭代）。
- `crate`：外部 crate 链接。详见crate 和模块（导入外部 crate）。
- `else`：`if` 和 `if let` 的失败分支。详见`if`，`if let`。
- `enum`：定义枚举。详见枚举。
- `extern`：外部 crate，函数，和变量链接。详见crate 和模块（导入外部 crate），外部函数接口。
- `false`：布尔值 false 常量。详见原生类型（布尔型）。
- `fn`：函数定义和函数指针类型。详见函数。
- `for`：循环，`impl` trait 语法的一部分，和高级生命周期语法。详见循环（for），方法语法。
- `if`：条件分支。详见`if`，`if let`。
- `impl`：继承和 trait 实现块。详见方法语法。
- `in`：`for` 循环语法的一部分。详见循环（for）。
- `let`：变量绑定。详见变量绑定。
- `loop`：无条件的无限循环。详见循环（loop）。
- `match`：模式匹配。详见匹配。
- `mod`：模块声明。详见crate 和模块（定义模块）。
- `move`：闭包语法的一部分。详见闭包（`move` 闭包）。
- `mut`：表示指针类型和模式绑定的可变性。详见可变性。
- `pub`：表示 `struct` 字段，`impl` 块和模块的共有可见性。详见crate 和模块（导出共有接口）。
- `ref`：通过引用绑定。详见模式（`ref` 和 `ref mut`）。
- `return`：从函数返回。详见函数（提前返回）。
- `Self`：（trait）实现者类型的别名。详见Traits。

- `self` : 方法的主体。详见方法语法（方法调用）。
- `static` : 全局变量。详见 `const` 和 `static` （ `static` ）。
- `struct` : 结构体定义。详见结构体。
- `trait` : trait 定义。详见Traits。
- `true` : 布尔值 true 常量。详见原生类型（布尔型）。
- `type` : 类型别名和关联类型定义。详见 `type` 别名，关联类型。
- `unsafe` : 表示不安全代码，函数，trait 和其实现。详见不安全代码。
- `use` : 向作用域导入符号。详见crate 和模块（使用 `use` 从模块导入）。
- `where` : 类型限制从句。详见Traits（ `where` 从句）。
- `while` : 条件循环。详见循环（ `while` ）。

运算符和记号

- `!` (`ident!(…)` , `ident!{…}` , `ident![…]`): 表示宏扩展。详见宏。
- `!` (`!expr`): 位计算或逻辑互补。可重载（ `Not` ）。
- `!=` (`var != expr`): 不等。可重载（ `PartialEq` ）。
- `%` (`expr % expr`): 算数取余。可重载（ `Rem` ）。
- `%=` (`var %= expr`): 算数取余并赋值。可重载（ `RemAssign` ）。
- `&` (`expr & expr`): 位计算和。可重载（ `BitAnd` ）。
- `&` (`&expr` , `&mut expr`): 借用。详见引用和借用。
- `&` (`&type` , `&mut type` , `&'a type` , `&'a mut type`): 借用指针类型。详见引用和借用。
- `&=` (`var &= expr`): 位计算和并赋值。可重载（ `BitAndAssign` ）。
- `&&` (`expr && expr`): 逻辑和。
- `*` (`expr * expr`): 算数乘法。可重载（ `Mul` ）。
- `*` (`*expr`): 解引用。
- `*` (`*const type` , `*mut type`): 裸指针。详见裸指针。
- `*=` (`var *= expr`): 算数乘法并赋值。可重载（ `MulAssign` ）。
- `+` (`expr + expr`): 算数加法。可重载（ `Add` ）。
- `+` (`trait + trait` , `'a + trait`): 复合类型限制。详见Traits（多个 trait bound）。
- `+=` (`var += expr`): 算数加法并赋值。可重载（ `AddAssign` ）。
- `,` : 参数和元素分隔符。详见属性，函数，结构体，泛型，匹配，闭包和crate 和模块（使用 `use` 从模块导入）。
- `-` (`expr - expr`): 算数减法。可重载（ `Sub` ）。

- `-` (`- expr`): 算数取反。可重载（ `Neg` ）。
- `-=` (`var -= expr`): 算数减法并赋值。可重载（ `SubAssign` ）。
- `->` (`fn(…) -> type` , `|…| -> type`): 函数和闭包的返回值类型。详见函数，闭包。
- `.` (`expr.ident`): 访问方法。详见结构体，方法语法。
- `..` (`..` , `expr..` , `..expr` , `expr..expr`): 右开区间的范围常量
- `..` (`..expr`): 结构体常量更新语法。详见结构体（更新语法）。
- `..` (`variant(x, ..)` , `struct_type { x, .. }`): “余下的”模式绑定。详见模式（忽略绑定）。
- `...` (`expr ... expr`): 闭区间范围模式。详见模式（范围）。
- `/` (`expr / expr`): 算数除法。可重载（ `Div` ）。
- `/=` (`var /= expr`): 算数除法并赋值。可重载（ `DivAssign` ）。
- `:` (`pat: type` , `ident: type`): 限制。详见变量绑定，函数，
- `:` (`ident: expr`): 结构体字段初始化。详见结构体。
- `:` (`'a: loop {…}`): 循环标签。详见循环（循环标签）
- `;` : 语句和项终结符。
- `;` (`[…; len]`): 定长数组语法的一部分。详见原生类型（数组）。
- `<<` (`expr << expr`): 左移。可重载（ `Shl` ）。
- `<<=` (`var <<= expr`): 左移并赋值。可重载（ `ShlAssign` ）。
- `<` (`expr < expr`): 小于。可重载（ `PartialOrd` ）。
- `<=` (`var <= expr`): 小于。可重载（ `PartialOrd` ）。
- `=` (`var = expr` , `ident = type`): 赋值 / 等价。详见变量绑定， `type` 别名，默认泛型参数。
- `==` (`var == expr`): 相等。可重载（ `PartialEq` ）。
- `=>` (`pat => expr`): 匹配分支语法的一部分。详见匹配。
- `>` (`expr > expr`): 大于。可重载（ `PartialOrd` ）。
- `>=` (`var >= expr`): 大于。可重载（ `PartialOrd` ）。
- `>>` (`expr >> expr`): 右移。可重载（ `Shr` ）。
- `>>=` (`var >>= expr`): 右移并赋值。可重载（ `ShrAssign` ）。
- `@` (`ident @ pat`): 模式绑定。详见模式（绑定）。
- `^` (`expr ^ expr`): 位计算异或。可重载（ `BitXor` ）。
- `^=` (`var ^= expr`): 位计算异或并赋值。
- `|` (`expr | expr`): 位计算或。可重载（ `BitOr` ）。
- `|` (`pat | pat`): 另外的模式。详见模式（多个模式）。
- `|` (`|…| expr`): 闭包。详见闭包。
- `|=` (`var |= expr`): 位计算或并赋值。可重载（ `BitOrAssign` ）。

- `||` (`expr || expr`): 逻辑或。
- `_` :“忽略”的模式匹配。详见模式（忽略绑定）。也被用来增强整型常量的可读性。
- `?` (`expr?`): Error propagation。当遇到 `Err(_)` 时提早返回，否则不执行，类似于 `try!` macro。

其他语法

- `'ident` : 命名的生命周期或循环标签。详见模式（绑定）。
- `…u8` , `…i32` , `…f64` , `…usize` , …: 特定类型的数字常量。
- `"…"` : 字符串常量。详见字符串。
- `r"…"` , `r#"…"#` , `r##"…"##` , …: 原始字符串常量，转义字符不会被处理。详见参考手册（原始字符串常量）。
- `b"…"` : 字节字符串常量，生成一个 `[u8]` 而不是一个字符串。详见参考手册（字节字符串常量）。
- `br"…"` , `br#"…"#` , `br##"…"##` , …: 原始字节字符串常量，原始和字节字符串常量的组合。详见参考手册（原始字节字符串常量）。
- `'…'` : 字符常量。详见原生类型（ `char` ）。
- `b'…'` : ASCII 字节常量。
- `|…| expr` : 闭包。详见闭包。

- `ident::ident` : 路径。详见crate 和模块（定义模块）。
- `::path` : 相对 crate 根的路径（也就是说，一个明确的绝对路径）。详见crate 和模块（ `pub use` 重导出）。
- `self::path` : 相对当前模块的路径（也就是说，一个明确的相对路径）。详见crate 和模块（ `pub use` 重导出）。
- `super::path` : 相对当前模块父模块的路径。详见crate 和模块（ `pub use` 重导出）。
- `type::ident` : 关联常量，函数和类型。详见关联类型。
- `<type>::…` : 一个不能直接命名的类型的关联项（例如， `<&T>::…` ， `<[T]>::…` 等）。详见关联类型。

- `path<…>` (*e.g.* `Vec<u8>`): 用类型指定泛型的参数类型。详见泛型。
- `path::<…>` , `method::<…>` (*e.g.* `"42".parse::<i32>()`): 用表达式指定泛型类型，函数或方法的参数。
- `fn ident<…> …` : 定义泛型函数。详见泛型。

- `struct ident<…> …` : 定义泛型结构体。详见泛型。
- `enum ident<…> …` : 定义泛型枚举。详见泛型。
- `impl<…> …` : 定义泛型实现。
- `for<…> type` : 高级生命周期 bound。
- `type<ident=type>` (*e.g.* `Iterator<Item=T>`): 一个泛型类型，它有一个或多个有特定赋值的关联类型。详见关联类型。

- `T: U` : 泛型参数 `T` 被限制为实现了 `U` 的类型。详见Traits。
- `T: 'a` : 泛型类型 `T` 必须超过声明周期 `'a` 。当我们说一个类型"超出"它的作用域时，意味着它不能间接的包含短于 `'a` 作用域的任何引用。
- `T : 'static` : 泛型类型 `T` 不包含除 `'static` 之外的被借用的引用。
- `'b: 'a` : 泛型生命周期 `'b` 必须超过声明周期 `'a`
- `T: ?Sized` : 允许泛型类型是一个不定长度类型。详见不定长类型。
- `'a + trait` , `trait + trait` : 复合类型限制。详见Traits（多个 trait bound）

- `#[meta]` : 外部属性。详见属性。
- `#![meta]` : 内部属性。详见属性。
- `$ident` : 宏替代（部分）。详见宏。
- `$ident:kind` : 宏 capture。详见宏。
- `$(…)…` : 宏重复（部分）。详见宏。

- `//` : 行注释。详见注释。
- `//!` : 内部行文档注释。详见注释。
- `///` : 外部行文档注释。详见注释。
- `/*…*/` : 块注释。详见注释。
- `/*!…*/` : 内部块文档注释。详见注释。
- `/**…*/` : 内部块文档注释。详见注释。

- `!` : 一个空的 Never type。详见发散函数

- `()` : 空元组（也就是单元），常量和类型。
- `(expr)` : 自带括号的表达式。
- `(expr,)` : 单元素元组表达式。详见原生类型（元组）。
- `(type,)` : 单元素元组类型。详见原生类型（元组）。
- `(expr, …)` : 元组类型。详见原生类型（元组）。
- `(type, …)` : 元组类型。详见原生类型（元组）。
- `expr(expr, …)` : 函数调用表达式。也用于初始化元组 `struct` 和元

组 `enum` 变量。详见函数。

- `ident!(…)`, `ident!{…}`, `ident![…]`: 宏调用。详见宏。
- `expr.0`, `expr.1`, ...: 元组索引。详见原生类型（元组索引）。
- `{…}`: 表达式块
- `Type {…}`: `struct` 常量。详见结构体。
- `[…]`: 数组常量。详见原生类型（数组）。
- `[expr; len]`: 包含 `expr` 的 `len` 次拷贝的数组常量。详见原生类型（数组）。
- `[type; len]`: 包含 `len` 个 `type` 实例的数组类型。详见原生类型（数组）。
- `expr[expr]`: 集合索引。可重载（`Index`，`IndexMut`）。
- `expr[..]`, `expr[a..]`, `expr[..b]`, `expr[a..b]`: 用来生成集合切片的集合索引，分别使用 `Range`，`RangeFrom`，`RangeTo`，`RangeFull` 作为“索引”。

参考文献

> bibliography.md
> commit 090fcb88b1c7353e480ce3aff0028832e9bb696b

这是一个与 Rust 相关的材料的阅读列表。这包含了曾经影响过 Rust 先验研究，以及关于 Rust 的出版物。

> （注：以下翻译属个人理解，勿作为参考！）

类型系统

- Cyclone语言中基于区域的内存管理（Region based memory management in Cyclone）
- Cyclone语言中的手动安全内存管理（Safe manual memory management in Cyclone）
- 类型类：使临时多态不再临时（Typeclasses: making ad-hoc polymorphism less ad hoc）
- 宏综述（Macros that work together）
- 特性：组合类型的行为（Traits: composable units of behavior）
- 消除别名（Alias burying） - 我们尝试了一些相似的内容并放弃了它
- 外部唯一性是足够的（External uniqueness is unique enough）
- 用于安全并行的唯一性和引用不可变性（Uniqueness and Reference Immutability for Safe Parallelism）
- 基于区域的内存管理（Region Based Memory Management）

并发

- Singularity：软件栈的重新思考（Singularity: rethinking the software stack）
- Singularity操作系统中支持快速和可靠的消息传递的语言（Language support for fast and reliable message passing in singularity OS）
- 通过work stealing来安排多线程计算（Scheduling multithreaded computations by work stealing）
- 多道程序多处理器的线程调度（Thread scheduling for multiprogramming

multiprocessors）
- work stealing中的数据局部性（The data locality of work stealing）
- 动态环形work stealing双端队列（Dynamic circular work stealing deque） - Chase/Lev双端队列
- 异步-完成并行的work优先和help优先的调度策略（Work-first and help-first scheduling policies for async-finish task parallelism） - 比严格的work stealing更宽泛
- 一个Java的fork/join灾难（A Java fork/join calamity） - 对Java fork/join库的批判，特别是其在非严格计算时的work stealing实现
- 并发系统的调度技巧（Scheduling techniques for concurrent systems）
- 竞争启发调度（Contention aware scheduling）
- 时间共享多核系统的平衡work stealing（Balanced work stealing for time-sharing multicores）
- 三层蛋糕？（Three layer cake）
- 非阻塞半work stealing队列（Non-blocking steal-half work queues）
- Reagents：表现和编写细粒度的并发（Reagents: expressing and composing fine-grained concurrency）
- 用于共享内存多处理器的可扩展同步性的算法（Algorithms for scalable synchronization of shared-memory multiprocessors）
- Epoch-based reclamation.

其它

- 只能崩溃的软件（Crash-only software）
- 编写高性能内存分配器（Composing High-Performance Memory Allocators）
- 对手动内存分配的思考（Reconsidering Custom Memory Allocation）

关于 **Rust** 的论文

- Rust中的GPU编程（GPU programming in Rust）
- 并行闭包：一个基于老观点的新做法（Parallel closures: a new twist on an old idea） - 并不完全关于Rust，不过是Nicholas D. Matsakis写的
- Patina: A Formalization of the Rust Programming Language。一类型系统子集的早期形式，Eric Reed著。
- Experience Report: Developing the Servo Web Browser Engine using Rust。Lars

Bergstrom著。
- Implementing a Generic Radix Trie in Rust。Michael Sproul的毕业论文。
- Reenix: Implementing a Unix-Like Operating System in Rust。Alex Light的毕业论文。
- Evaluation of performance and productivity metrics of potential programming languages in the HPC environment。Florian Wilkens的学士学位论文。比较C，Go和Rust。
- Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust。Geoffroy Couprie著，关于VLC的研究。
- Graph-Based Higher-Order Intermediate Representation。一个用Impala（一个类似Rust的语言）实现的实验性的IR。
- Code Refinement of Stencil Codes。另一个使用Impala的论文。
- Parallelization in Rust with fork-join and friends. Linus Farnstrand's master's thesis.
- Session Types for Rust. Philip Munksgaard's master's thesis. Research for Servo.
- Ownership is Theft: Experiences Building an Embedded OS in Rust - Amit Levy, et. al.
- You can't spell trust without Rust. Alexis Beingessner's master's thesis.

名词中英文对照

> 这里收集了一些 Rust 的名词翻译，并在此统一说明。如果你在阅读时遇到无法理解地方，请提 issue。

English name	中文名	备注
attribute	特性	
coercion	强制转换	
composition	组合	
deref coercions	解引用强制多态	
expansion / expand	展开	
extension / extend	扩展	
free variables	自由变量	
idempotent	幂等	
noalias	不译	http://llvm.org/docs/LangRef.html#noalias
aliasing	重叠	http://llvm.org/docs/LangRef.html#pointer-aliasing-rules

slice	不译	`slice` 不做翻译，文档中的“切片”都是 `slice` 的意思。
synchronous types	同步类型	
trait	不译	`trait` 不做翻译
trait object	trait 对象	
undefined behavior	未定义行为	
unsafe	不安全	
vector	不译	鉴于将 `vector` 翻译为 `向量` 容易引起误解，故决定不再对其进行翻译，如果你在本书中看到“向量”一词，这一定是还未修改过来，请自行脑补为 `vector`